Practical Computing

For Science and Engineering

Bryan Wright

2023

BRYAN WRIGHT

PRACTICAL COMPUTING

FOR SCIENCE AND ENGINEERING

PHYSICS DEPARTMENT
UNIVERSITY OF VIRGINIA

Version 0.51, August 2022

PHYSICS DEPARTMENT
UNIVERSITY OF VIRGINIA

Contents

For Finian

Introduction

A New Kind of Problem-Solving

It's a lazy summer afternoon in 1450, and a tired monk is sitting at a desk, staring at a blank sheet of vellum. He's been given the task of making twenty copies (twenty!) of a fifty-page book. He sighs, then picks up a pen and begins to write, following the holes that have been carefully pricked into the sheet as guides. He wonders if he's being punished. This will take forever!

As he works, his mind wanders into fantasies of being an Abbott or a King, capable of commanding monks to do all the menial work. He'd only have to command twenty copies of a book (or a thousand!) and it would be done. Even better to be a Wizard, and not have to deal with lazy monks! Swoosh! goes the magic wand, and a pile of books appears!

Figure 1: A monk copying a manuscript.
Source: Wikimedia Commons

The monk doesn't know it, but his vision is becoming reality even as he works. A few years earlier, Johannes Gutenberg had invented a printing press that used moveable type. As it spread across Europe, this new technology was changing the way people thought about problem-solving.

For the monk in his scriptorium, each new page is a new problem requiring an amount of time and effort similar to any previous page. To copy fifty pages takes him about fifty times as long as a single page. Even though he might begin the task by spending a little time thinking about the style of the writing and the layout of the pages, the vast majority of his time will be spent on the mindless, repetitive task of producing individual pages, one at a time. If his mind wanders into fantasies, a page could be ruined.

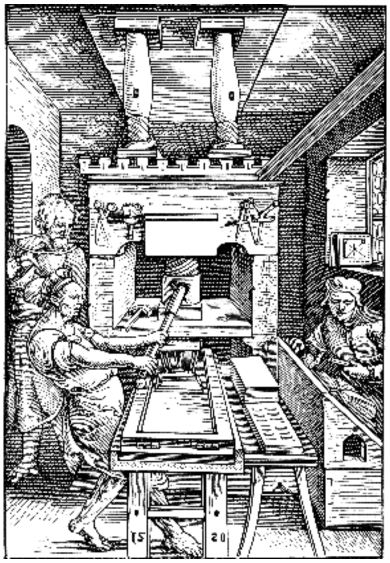


Figure 2: A printing press (1520).
Source: Wikimedia Commons

But consider the job of a printer a hundred years later. To him, the problem of printing a page consists of setting the type. Once he's done that, he can create as many copies of the page as he likes, with relatively little effort and in a short time.

* * *

Early 20th Century particle physicists used "cloud chambers" and, later, "bubble chambers" to see the paths of subatomic particles. Collisions and decays within these chambers produced visible tracks that could be photographed. The chambers could take a new photograph every few seconds. Each photograph was then analyzed by people called "scanners", who measured the tracks as the photographs were projected onto a table. At their fastest these workers could analyze only about five photographs per hour. Photographs taken during a few days of running a bubble chamber could take years to analyze.

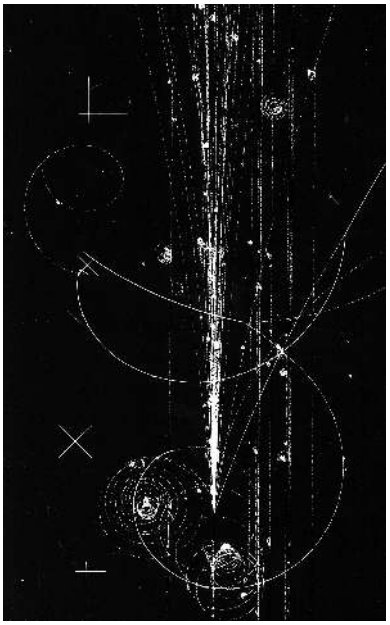


Figure 3: Traces of charged particles in a bubble chamber at Fermilab (1973).

Source: Wikimedia Commons

Bubble chambers have long been superseded by other kinds of detectors that can be read out electronically and analyzed by computers. Because of this, large experiments like the Compact Muon Solenoid at CERN can record and analyze thousands of electronic "snapshots" per second. There are no longer any "scanners", just as monks no longer copy manuscripts.

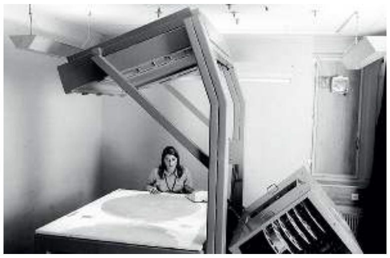


Figure 4: A "scanner" analyzes a bubble chamber photograph.

Source: CERN

* * *

Since the earliest days of aeronautics, airplane designs have been tested in wind tunnels. The Wright brothers themselves used a simple wind tunnel in the development of the "Wright Flyer". Whole airplanes, parts of them, or models of them were placed into the wind tunnel to study their behavior. The lift generated by one type of wing or propeller might be measured and compared to measured values for other designs. Many models were made and tested in the process of designing an airplane.

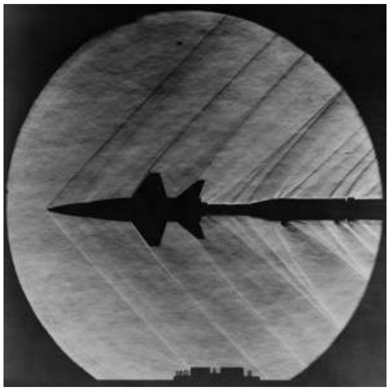


Figure 5: A model of the X-15 rocket plane in a wind tunnel (1962).

Source: Wikimedia Commons

Today, computer simulations have largely replaced wind tunnel tests. Modern computational fluid dynamics can accurately model the flow of air around complicated shapes, and we can change the shape by clicking and dragging a mouse or changing some parameters, rather than needing to manufacture a physical model, leaving the engineer free to test odd shapes and explore possibilities as they occur to her.

* * *

In 1913 Henry Norris Russell documented a relationship between the color and brightness of stars. At that time, and indeed until the 1970s, most graphs used in publications were drawn by hand. On the left-hand side of the figure below you can see Russell's graph of brightness versus color (what we now call a Hertzsprung-Russell diagram). The graph shows data for about 300 stars, collected by observers using astronomical instruments and written down by hand. These data were then plotted, using pen and ink, to show the results.

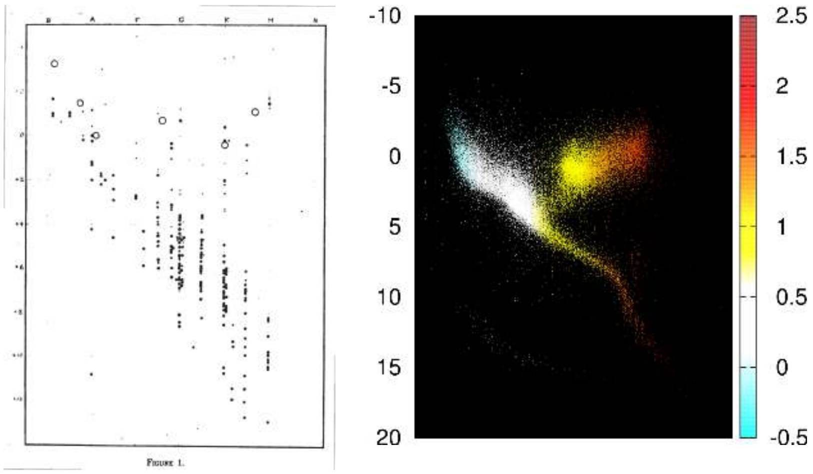


Figure 6: Russell's original diagram, and a modern Hertzsprung-Russell diagram produced with *gnuplot* using data from the Hipparcos satellite.

Source: Popular Astronomy. 22: 275-294, 1914

On the right-hand side of the figure above we see a modern-day Hertzsprung-Russell diagram. It was produced using data gathered by the Hipparcos satellite, downloaded over the Web, analyzed by a computer program, and plotted using *gnuplot*. It shows about 100,000 stars. It took the computer less than a second to produce this graph from the data.

* * *

The computer revolution of the late 20th Century gave us a new kind of problem-solving. As in the aftermath of the Gutenberg revolution, we suddenly found that we no longer needed to focus on the mindless, repetitive components of many tasks. Computers could now make data analysis more-or-less effortless. Simulations done by computers were now capable of eliminating the need for many real-world tests. Visualizations that were once tedious to prepare could now be done instantly, by anybody. The ease, accuracy, and speed with which computers could perform repetitive tasks freed us up to explore in ways that would have been unfeasible earlier.

To a poor monk in a scriptorium every page is a new problem that needs to be solved. To a printer, once the page is typeset the problem is solved forever. A well-written computer program does the same. It tackles a problem, and solves it *forever*. That's a new kind of problem-solving.

About this Book

Today, if you intend to pursue a career in science or engineering you'll need to know the basics of computation. This book aims to teach them to you.

It introduces three core skills: analyzing data, simulating data, and visualizing data. It assumes no prior programming experience or knowledge about the inner workings of computers. It will concentrate on using using computers to solve common problems you'll encounter in science and engineering.

A Note About Choices

Which is the best tool: a hammer or a screwdriver? Most people would say that the answer depends on the task. The same is true for computer languages. There is no "best" programming language, any more than there's a best tool.

When designing this book, I needed to choose a programming language that would suit its needs and yours. I settled on the C language for several reasons.

First of all, C and its cousins (C++, Objective-C, *etc cetera*) are very widely used. It's likely that any program you've ever used on a desktop computer was written in some variant of the C language. A 2016 study by IEEE[1]ranked C as the most popular programming language, based on its use in software repositories and appearance as a topic in various online forums.

C has been around a long time, and many newer programming languages have adopted features from it. This means that once you've learned C you'll find it easier to learn those languages, too. Some of these C-like languages include Java, PHP, Javascript, Perl, Go, and C#.

More than some languages, C lets you see the computer's internal workings. When learning C, you need to think about the way the computer uses memory to store information, and how data is stored in

[1] IEEE Spectrum: The Top Programming Languages 2016

files. An understanding of these concepts will help you later on, even if you move to higher-level programming languages that hide these details.

C has a reputation for being fast. Other languages sometimes rely on C to do their "heavy lifting". For example, Google recently released an artificial intelligence system named TensorFlow[2], which appears to be written in the Python programming language. If you download TensorFlow and look at the source code, though, you'll find that about 80% of it is written in C. The Google developers said they wrote the most compute-intensive parts of the code in C to make it run faster. If you go into research or engineering, you'll often be working at the cutting edge of technology. Having the skill to write C programs can help you squeeze the best performance out of your software.

Finally C is available on a wider range of computers than any other language, and the software needed to build C programs is available for free. No matter what kind of computer you're using, or how small your budget, It's almost certain that you'll be able to write and run C programs.

Figure 7: Dennis Ritchie, the inventor of the C language.
Source: Wikimedia Commons

Those are some of the reasons for choosing to use the C language in this book. Every language has its strengths and weaknesses. After you've learned C, I hope you go on to explore other languages too. When you're a researcher or an engineer, here are some other things you should think about when deciding which language to use for a project:

- What are your skills? Sometimes its better to use a language you already know.
- What are the skills of other programmers who are likely to work on this project in the future? When you're collaborating with other programmers, consider their skills, too.
- If there's an existing code base, what language(s) does it use? When adding features to existing software, it's often a good idea to stick to the same language the rest of the software uses, unless there's a compelling reason to introduce a new language.
- Are strengths of a given programming language a good match for the project's needs? Don't try to use a hammer to insert screws.

[2] https://www.tensorflow.org/

1. Zero to Loops

1.1. What's a Program?

Computers today do a lot of complicated things, from weather prediction to playing music, movies and games.

You might be surprised to learn that computers have been around since ancient times. One early computer was the "Antikythera Mechanism", found in a 2,000-year-old Greek shipwreck. This complicated machine could be used to predict the future positions of astronomical bodies and the phases of the moon.

The Antikythera Mechanism.

Source: Wikimedia Commons

The Antikythera Mechanism did many things, but unlike modern computers it wasn't possible to add new capabilities after the machine was made. All of its capabilities were determined when it was built. If someone needed to do something that it wasn't built to do, they'd need to buy or build a new device with different capabilities.

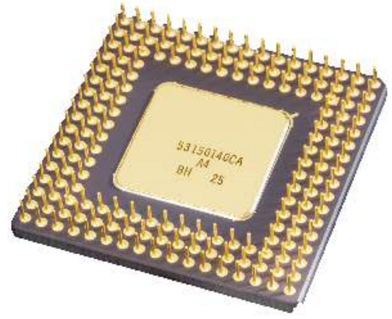
In the early 1800s, the English scientist and engineer Charles Babbage proposed a new kind of computer that he called an "Analytical Engine". This would be a general-purpose computer. Its behavior was controlled by punched cards (rectangular cards with a pattern of holes in them). By creating an appropriate set of cards, the Analytical Engine could be made to do *any* calculation. (Similar punched cards had previously been used to control the patterns woven into fabric by looms.) The mathematician Ada Lovelace, working with Babbage, created the first sets of cards for this versatile early computer.

Ada Lovelace, the first computer programmer.

Source: Wikimedia Commons

Most modern computers are designed to be versatile: a given computer can be used to do many different things. We add new abilities to the computer by installing "programs" into the computer.

We distinguish between the computer's "hardware", which is fixed and unchangeable, and its "software", which can be easily changed. Computer programs are part of the computer's software. Examples of computer programs you're probably familiar with include Firefox, Safari, Excel, Word, PowerPoint, PhotoShop, and many others.



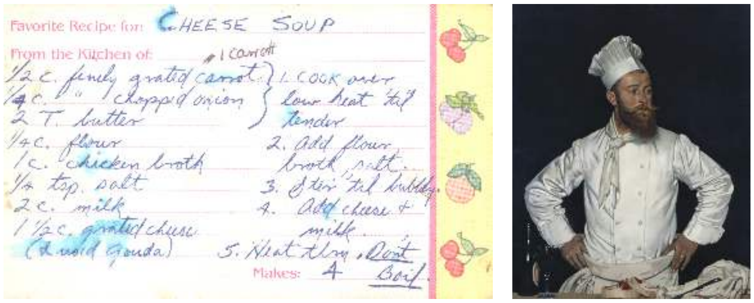
An Intel 80486 CPU. In general, different brands and models of CPU understand different sets of instructions, but most processors used today share a common set of core instructions that they all understand.

Source: Wikimedia Commons

1.2. Creating Programs

How can we create a program that tells a computer what we want it to do?

If the computer were a chef, we could tell it how to make our favorite dish by writing down a recipe. There's a problem, though: the chef in this case (the computer) doesn't speak English.



Favorite Recipe for: CHEESE SOUP
From the Kitchen of: 1 carrot

1/2 c. finely grated carrot
1/4 c. " chopped onion
2 T. butter
1/4 c. flour
1 c. chicken broth
1/4 tsp. salt
2 c. milk
1 1/2 c. grated cheese
(a mild Gouda)

1. COOK over low heat til tender
2. Add flour, broth, salt.
3. Stir til bubbly.
4. Add cheese + milk.
5. Heat thru, Don't Boil.

Makes: 4

Figure 1.1: A program is just a recipe, but it needs to be translated into a language the computer can understand.

Source: Wikimedia Commons 1, 2

The computer's brain is a "Central Processing Unit" (CPU), often just called a "processor". It only understands instructions that are expressed in a language of binary numbers.

A binary number is a number written in base 2. All of the digits of such a number are either zeros or ones, like this: 10110010. You can think of a binary number as a line of switches that can be turned on or off. (See Figure 1.2.)

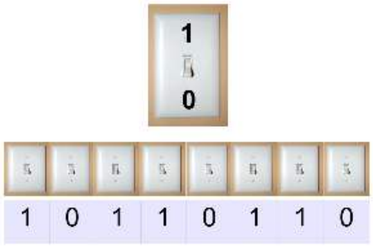


Figure 1.2: Bits as switches.
You can think of each bit in a binary number as as switch. (In fact, programmers often talk about flipping bits on or off.) We group bits together in groups of eight because eight is a power of two (2^3), making it convenient for binary (base-2) arithmetic, just as 10, 100 or 1000 are convenient in base-10. The very popular early Intel CPUs used data in 8-bit chunks, and this became a *de facto* standard.

Each digit of a binary number is called a "bit".[1] We say that a bit is either "on" or "off" (1 or 0). We usually group bits together in sets of eight. A set of eight bits is called a "byte".

Although it's possible to create a computer program by writing long streams of bits by hand, it's really tedious and prone to error. Even a moderately-sized program is millions of bytes long.

What we need is some kind of translator who can read a recipe in a language that's easy for us to write, and then translate it into the binary language that the computer understands.

[1] Some people claim that "bit" is a shortened form of "binary digit", but I'm skeptical.

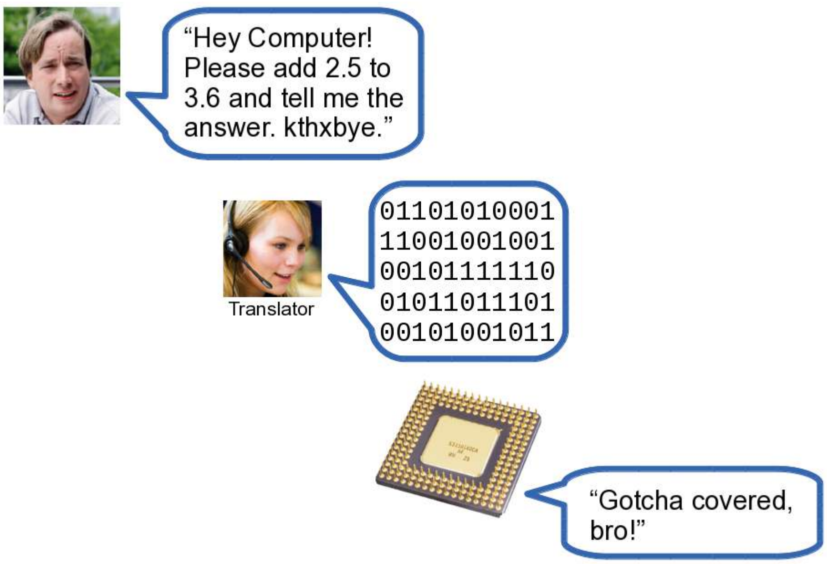


Figure 1.3: *Source: Wikimedia Commons 1, 2, 3*

The kind of translator we’ll be using in this book is called a *compiler*. It takes a readable description of what we want the computer to do (our “recipe”) and translates it into binary instructions.

We can’t quite write our program’s “recipe” in a human language like English, but there are many programming languages that have been developed to be readable by humans but still express our wishes in a clear, simple way that can easily be translated into the computer’s native binary language.

One of the most widely used programming languages is called simply “C”. That’s the language we’ll be using in this book.[2] The vast majority of the software you’ve used is written in C, or its cousin C++. You’d be hard-pressed to name a piece of software on your computer, phone or tablet that wasn’t written in C or one of its close relatives.

Think of the C language as a very terse version of English, with some special characters to help make your meaning clear. You might compare it to text messages or e-mails.

Program 1.1 is a simple program written in the C language:

Program 1.1: hello.cpp

```
#include <stdio.h>
int main () {
      printf ( "Hello World!\n" );
}
```

[2] There are hundreds of different computer languages. Each has its own strengths and weaknesses, and no language is best for all tasks. When choosing a language for a particular project, programmers think about whether the language’s strengths are a good match for that project.

This program just prints out the text “Hello World!”. Don’t worry about understanding it right now. We’ll explain how it works soon.[3]

At this point there are three obvious questions:

- Where do we type these instructions?
- How do we get a compiler to translate them into binary instructions that the computer can use?
- How do we get the computer to run the program we’ve created?

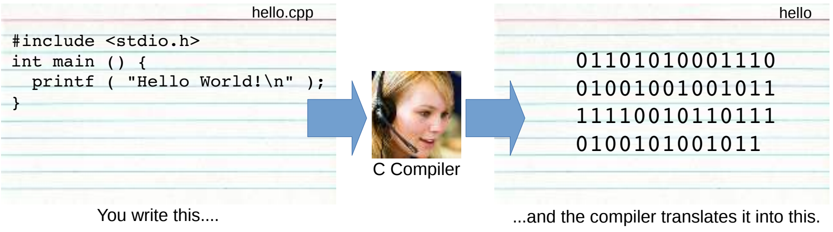
Before we can answer these questions, there’s one more thing we need to talk about: *files!*

1.3. Files

Before the compiler can translate your recipe, it needs to be written down. Instead of using pencil and paper, you’ll be writing your recipe into a *file* that lives on the computer’s hard disk. A file is just a named bunch of data. You can think of it as an index card with some information scribbled on it, and a title (the file’s name) written at the top.

Here’s how to create a program: First, we use a piece of software called an *editor* (this is our “pencil”) to create a file that contains some directions written in the C language (our “recipe”)[4]. Then we use a piece of software called a *C Compiler*. The compiler reads the file we’ve created and makes a binary version of our instructions in a new file[5]. The new file is our program, and we can run it just like any other program on the computer.

This binary file is a new piece of software that we’ve created. If we were a software company like Microsoft, we could sell this binary file to our customers, and they could put it onto their computers and use it.



```
#include <stdio.h>
int main () {
  printf ( "Hello World!\n" );
}
```

```
01101010001110
01001001001011
11110010110111
0100101001011
```

Figure 1.4: The C compiler reads our source code file and makes a binary file that the computer can understand.

Source: *Wikimedia Commons*

[3] On Wikipedia you’ll find a long list of “Hello World” programs written in many different languages. Some of them are truly bizarre.

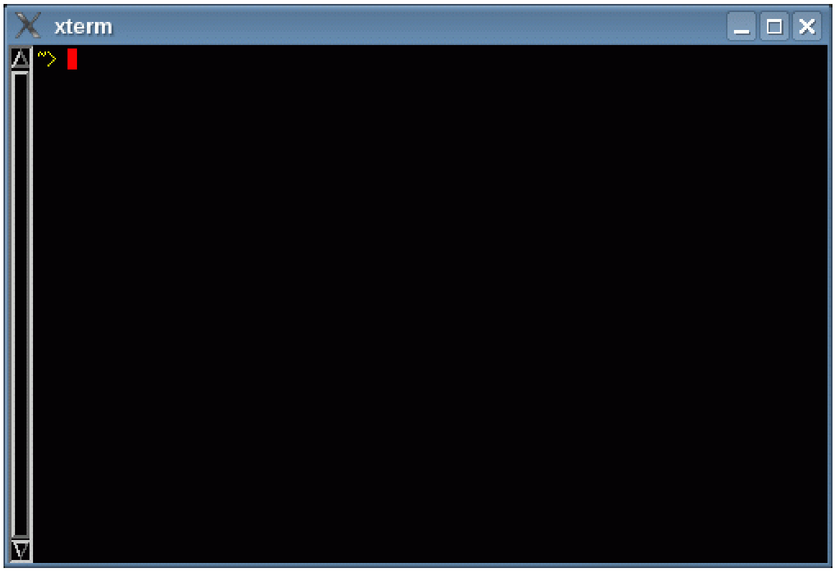
[4] This description is often called the program’s “source code”

[5] The binary file is often called an “executable” or just a “binary”

1.4. Your First Program

Let's look at the details of each of the steps in creating a program. In the following exercise we'll be creating the example program called `hello.cpp` (Program 1.1) that we saw earlier.

Most of our work will be done from the command line, so the first thing you'll need to do is open an appropriate *command window*. A command window is a box like the one shown in Figure 1.5. If you don't know how to open one, see Appendix B for instructions tailored to the kind of computer you're using (Windows, Mac, or Linux). You can tell your computer what to do by typing commands into this window.



```
~>
```

Figure 1.5: A command window. The appearance will vary, depending on what kind of computer you're using.

Writing a Program

To write our program, we'll use a piece of software called a *text editor*. It lets you type in some text, and save the text into a file. The text editor we'll be using is called *nano*.[6]

nano runs inside the command window. To create a file with nano, or modify an existing file, just type "`nano`" followed by the file name. Start it up now by typing "`nano hello.cpp`". Figure 1.6 shows what *nano* looks like while you're using it.

In *nano*, you can just type the text of your program. At the bottom of the window, you'll see that *nano* gives you some hints about how to do things. For example, you'll see that `^X` means "Exit". Here, `^X` means "hold down the Ctrl key while pressing the X key".

[6] You'll find instructions in Appendix B for installing *nano* and the other software you'll need for the exercises in this book.

Exercise 1: Creating a "Hello World" Program

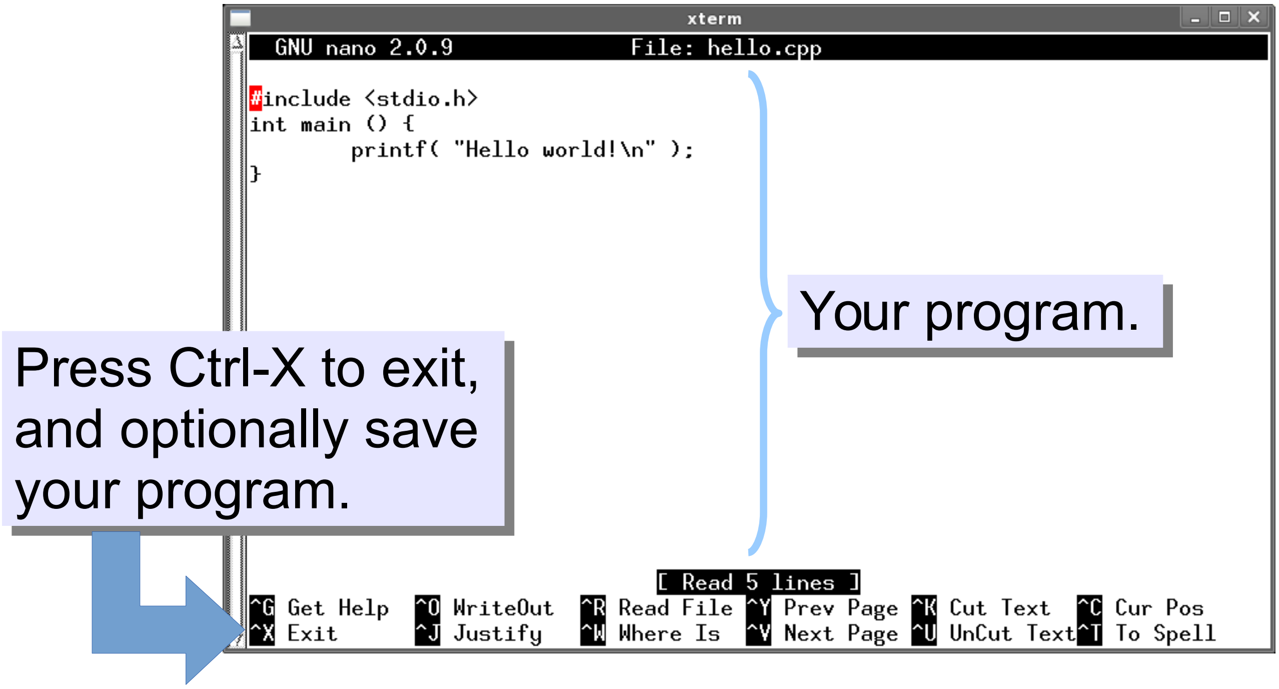
Start up *nano* and type the program "hello.cpp" that you saw earlier (Program 1.1, above). When you've finished typing, it should look like figure 1.6.

You should be careful to type the program exactly as it's written here. In particular, always remember that the C programming language cares about whether letters are upper- or lower-case. In C, the word "This" isn't the same as "this" or "THIS".

Once you've finished typing your program, save it by pressing `^X` (hold down the CTRL key, and press the X key). You'll be asked to confirm that you want to save your work into a file (type "y" for yes), and asked what you want to call the file. In response to this, type hello.cpp and then press enter. This creates a file called "hello.cpp", puts the things you've typed into it, and closes *nano*.

For best results when writing your own programs, stick to all lower-case unless there's a good reason to do otherwise.

You can see the new file you've created by typing the command "`ls`" (which is short for "list"). This will show a list of your files. You should see a file named "`hello.cpp`".



```
#include <stdio.h>
int main () {
        printf( "Hello world!\n" );
}
```

Your program.

Press Ctrl-X to exit, and optionally save your program.

Figure 1.6: The editor called "*nano*".

Compiling Your Program

Now we need to translate your program into binary instructions that the computer can understand.[7] We use a compiler to do this. The compiler we will use in this book is named *g++*. (This is pronounced "g plus plus".)

Exercise 2: Compiling "hello.cpp"

Use *g++* to compile your program by typing the following in your command window:

```
g++ -Wall -o hello hello.cpp
```

This tells *g++* to read the file `hello.cpp` and create a binary version of the program in a new file, named `hello`. Here's what the parts of the command mean:

"`-Wall`" means "Warn me if you see anything wrong with my program"

"`-o hello`" means "Write the output into a file named `hello`"

If you see any error messages, check to make sure you've typed the program correctly. In particular, look for missing semicolons and brackets, or places where you might have used parentheses instead of brackets. To look at your program again and fix any errors, just type "`nano hello.cpp`" again. When you're finished making changes, use `^X` as you did before to save your changes and exit from *nano*. Then try compiling your program again, as described above. Does it work now?

As you saw in the previous exercise, you can use the `ls` command to see a list of your files. If you do this now, you'll see that you've created a new file named `hello`.

[7] We call this "compiling the program".

Running your program

You've created the file hello.cpp, containing a "recipe" for making your program, and you've used *g++* to translate this into binary instructions the computer can understand, and write these instructions into the file hello. Now you're ready to run your program!

Exercise 3: Run it!

Tell the computer to run your program by typing the following command:

```
./hello
```

You should see the words "Hello World!". Congratulations! You're a programmer.

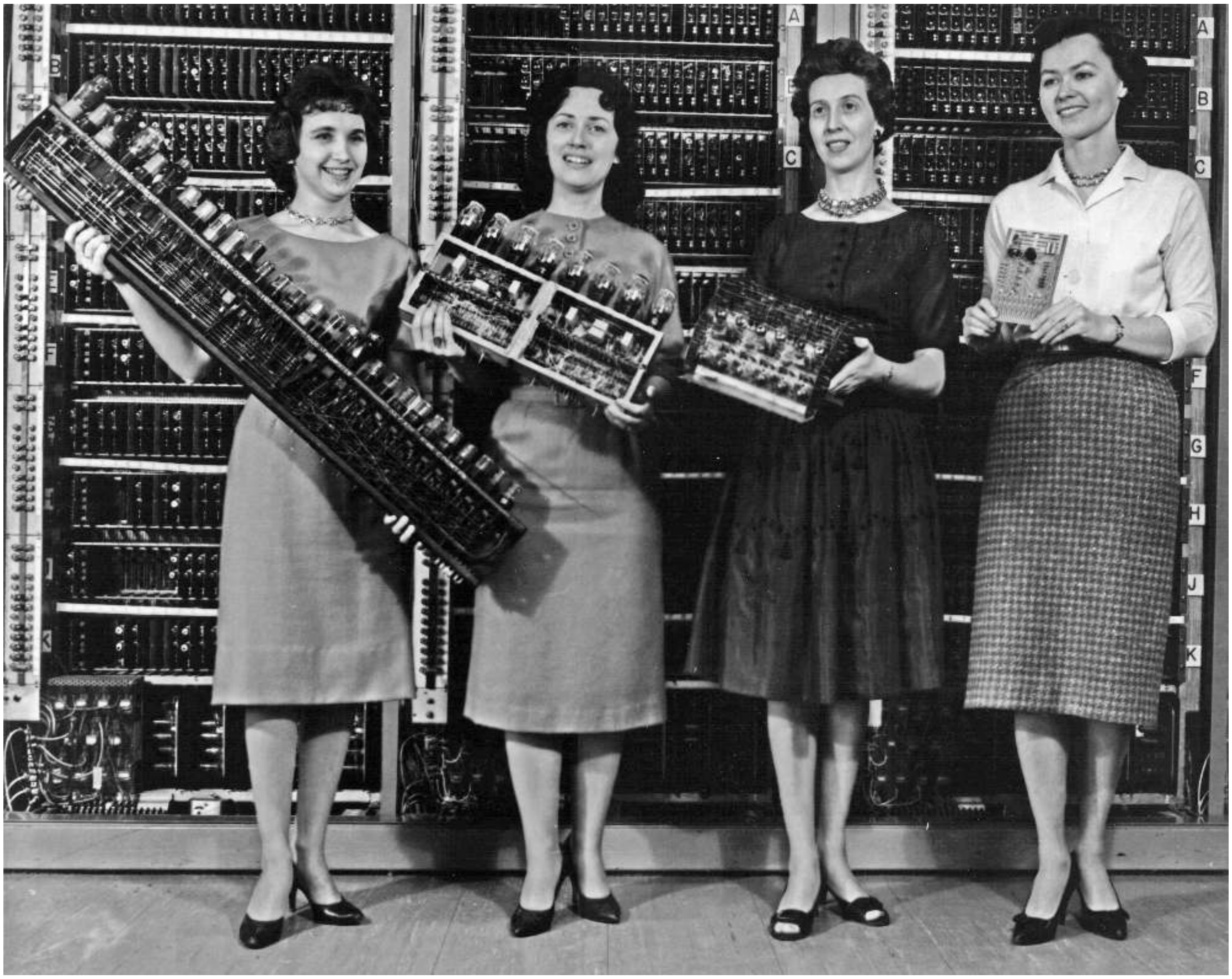
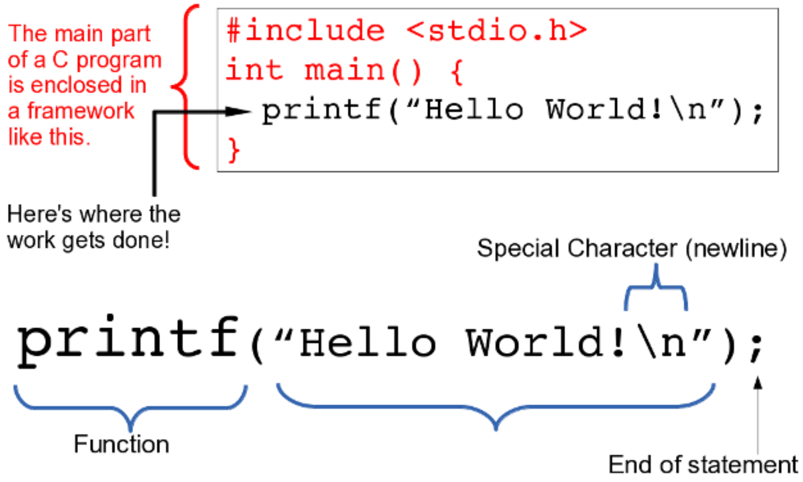


Figure 1.7: Congratulations!
Source: Wikimedia Commons

1.5. The Anatomy of a Program

What do the different parts of your simple C program do?



```
#include <stdio.h>
int main() {
   printf("Hello World!\n");
}
```

The main part of a C program is enclosed in a framework like this.
Here's where the work gets done!
Special Character (newline)
printf("Hello World!\n");
Function
End of statement

Figure 1.8: The anatomy of our "Hello World" program.

All but one line of this program is a framework that we'll use for most of the programs we write in this book. As you learn more you'll understand what each part of this framework does, but for now please just accept it as it is.

The one line of the program that is of immediate interest is the one that reads:

```
printf( "Hello World!\n" );
```

This is a single statement in the C language, and it tells the computer to write the text "`Hello World!`". The "\n" at the end tells the computer to go to the next line after it's written this text.[8]

What would happen if we left out the "\n"? It would be easier to see the effect of the "\n" if our program had two `printf` statements, like this:

```
printf ( "Hello World!\n" );
printf ( "...and Dog!\n" );
```

A program like this, when compiled and run, would print out:

```
Hello World!
...and Dog!
```

But if we left off the "\n" in the first `printf` statement the program

[8] "\n" means "insert a newline". As we go along, you'll see other similar things beginning with "\" and controlling how the computer writes text.

would print:

```
      Hello World!...and Dog!
```

See the difference?

`printf` itself is called a *function*. Just as functions in algebra may have arguments, so can C functions. In this case, we're giving the `printf` function one argument: the text to be printed. We'll see many more C functions as we go along.

Finally, at the end of our printf statement we see a semicolon. Why is it there? Because the C language allows us to write our statements on multiple lines if we want to. We could, for example, have written our printf statement like this:

```
     printf (
        "Hello World!\n"
     );
```

The semicolon at the end tells the C compiler that we're done with this statement now, and ready to go on to the next one. Think of the semicolon as being like the period at the end of a sentence.[9]

But what about...?

Could we write something like this?

```
     printf(
        "Hello
              World!\n"
     );
```

No, it turns out that this won't work. A broken chunk of quoted text like this will confuse the C compiler and cause it to refuse to compile our program.

If we really wanted to break the quoted text across two lines, we'd need to insert a "\" after "Hello", like this:

```
     printf(
        "Hello\
              World!\n"
     );
```

The "\" means "continued on next line". Note that there can't be

[9] Some other computer languages actually *do* use a period to indicate the end of a statement. (Cobol is one of these.) C doesn't use a period because it has another use for that, which we'll see later, in Chapter 12.

any spaces after the “\”, either.

This kind of thing is bad form, though, and shouldn’t be done in a real program unless there’s a compelling reason to do so. It just makes our program harder to read, and that’s usually a bad thing.

In fact, if we really wanted to make the program difficult to read, we could use the “\” to break up other things:

```
prin\
tf(
   "Hello\
         World!\n"
};
```

Now that’s hard to read! Don’t write programs this way. It’s icky!

1.6. Doing Math

Let’s try working with numbers now. Imagine I have $25.00 in my wallet and $238.00 in the bank. How much money do I have in total? Let’s ask the computer to do the math for us, like this:

```
printf ( "Total funds: %lf\n", 25.0+238.0 );
```

Notice that now we’re giving `printf` two arguments. The first argument is some quoted text, as before. But now we’ve added a second argument (separated from the first by a comma) that looks like an arithmetic expression. To understand what all of this does, we’ll first need to know a little more about how printf works.

The first argument given to `printf` will always be a chunk of quoted text. Sometimes this will be the *only* argument. In our “Hello World!” example, the only argument we gave to `printf` was the text that we wanted it to print.

In general, though, you can think of the text in this first argument as a fill-in form we give `printf`. (See Figure 1.9.) It can contain placeholders that mark spots where we want `printf` to figure something out, and fill in the blanks for us.

In the `printf` example above, the three characters `%lf` (percent, `l` as in “Lucy”, `f` as in “Fred”) together form a placeholder, marking a spot where the computer is supposed to insert a number. More specifically,

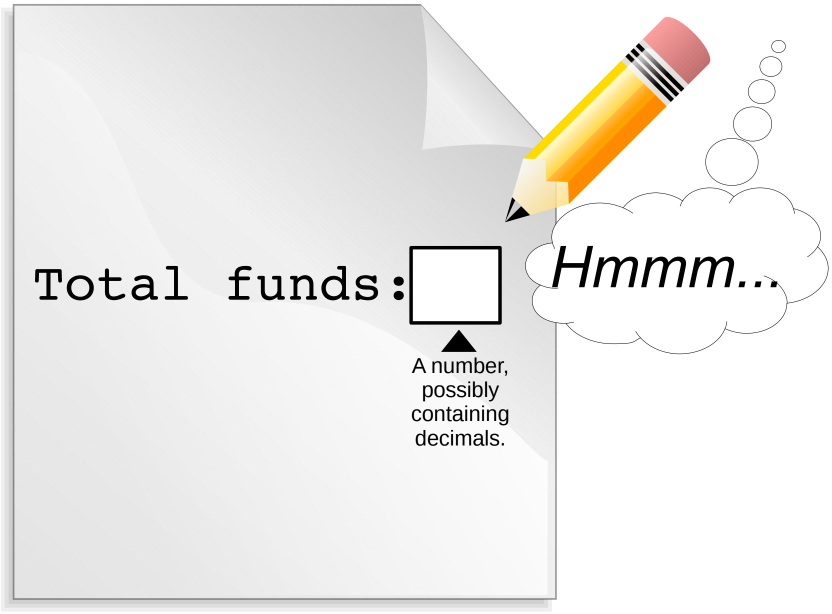


Figure 1.9: The text we give printf is like a fill-in form.

%lf means "save a spot here for a number that may contain decimal places"[10]. We'll encounter several other placeholders like this later, each of them for a different kind of number (or some other kind of thing we'd like to print out).

In our example, the second argument tells printf what we want to insert into the spot reserved by the placeholder. In this case, we give it the mathematical expression 25+238. The printf function will do the math for us, fill in the blank, and print out the result.

Let's look at a slightly less trivial problem (see Figure 1.10). Imagine we have a linear function, $y = 2x + 3$, and we want to know what the value of y will be when $x = 4.3$. How could we write a simple C program to tell us the answer?

Here's one way to do it (notice that the symbol for multiplication in C is an asterisk):

```
#include <stdio.h>
int main () {
  printf ( "The answer is %lf\n", 2.0 * 4.3 + 3.0 );
}
```

If you wrote this program, compiled it, and ran it, it would print out "The answer is 11.6", which is the correct value of y.[11]

[10] We'll discuss what the letters lf stand for a little later.

[11] Actually, you'll see that the program prints out something like "The answer is 11.600000". We'll see how to control how many decimal places are printed later.

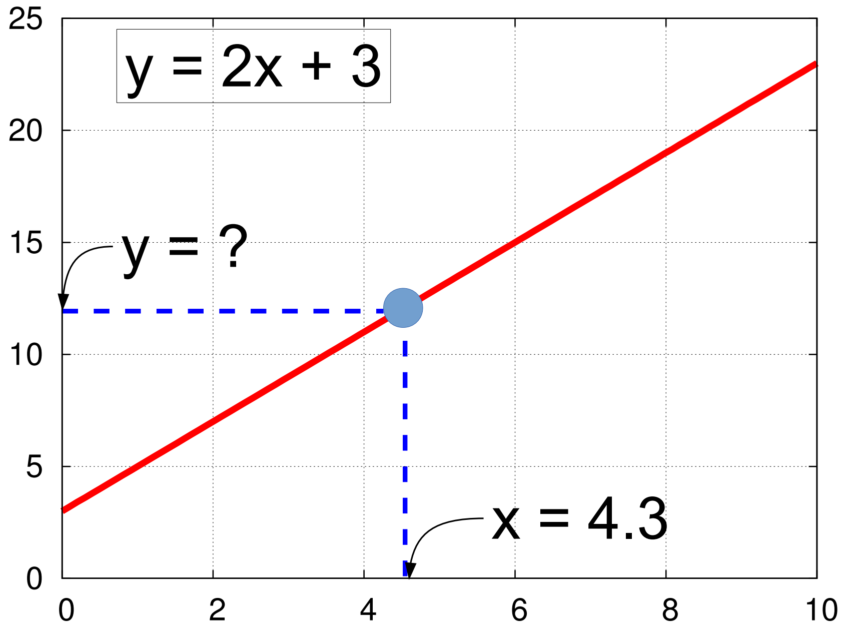


Figure 1.10: A line representing the equation y(x) = 2x + 3.

`printf` evaluates the mathematical expression `2.0 * 4.3 + 3.0` to get the value 11.6, and then inserts this number in place of `%lf`.

Placeholders like `%lf` are called *format specifiers*. They tell the computer where to insert something and how it should be formatted. We can use more than one format specifier to insert multiple numbers into the text. For example:

```
#include <stdio.h>
int main () {
   printf ( "At x=%lf the value of y is %lf\n",
             4.3,
             2.0 * 4.3 + 3.0 );
}
```

Note that I've broken the line up because it's long. This is OK, as long as I don't insert a line break in the middle of a word or a chunk of quoted text without using a "\" continuation character.

This program would print "`At x=4.3 the value of y is 11.6`". The first `%lf` gets replaced with the first number, and the second `%lf` gets replaced with the second number. (See Figure 1.11.)

1.7. Variables

When you look at the expression "`2.0 * 4.3 + 3.0`" do you remember what the numbers represent? Which is the line's slope? Which is the y-intercept? Which is the value of x? If we came back to this

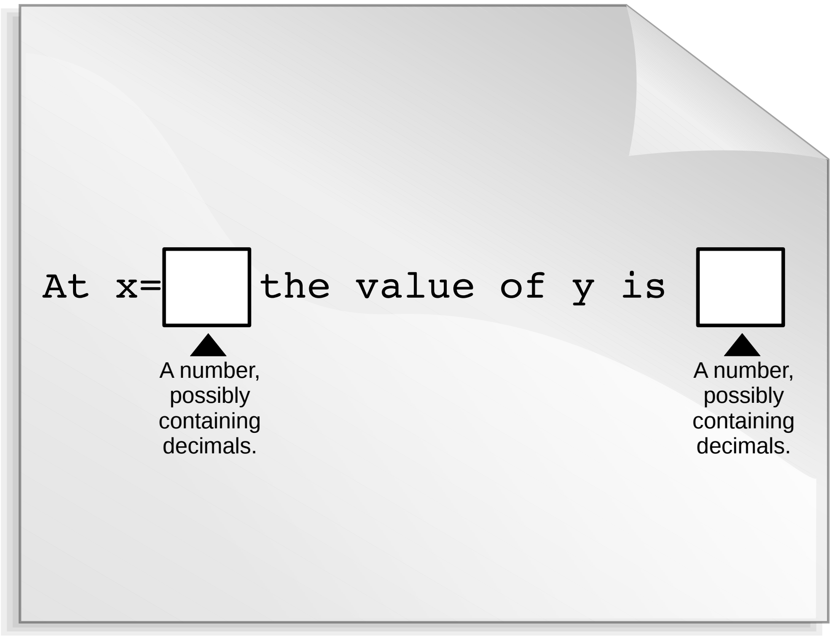

At x= □ the value of y is □
A number, possibly containing decimals.
A number, possibly containing decimals.

Figure 1.11: The `printf` text can contain more than one placeholder.

program later, we might not have any idea which number was which. Let's get organized!

Here's another version of the program:

```
#include <stdio.h>
int main () {

      double x;
      double y;
      double slope = 2.0;
      double yint  = 3.0;

      x = 4.3;
      y = slope * x + yint;

      printf ( "At x = %lf the value of y is %lf\n", x, y );
}
```

Definitions of Variables

Now our mathematical expression is "`slope * x + yint`", which should be much easier to understand.

We've defined four *variables* in this program: `x`, `y`, `slope`, and `yint`.

A variable is a named box into which we can put a value.[12] Variables in C are similar to variables in algebra, except that there are different kinds of C variables for holding different kinds of data.

The four lines beginning with the word `double` define the four variables we're going to use. "Defining" the variable means telling the computer what kind of values you'll assign to the variable. (In C, you must define variables before you can use them.) While you're defining the variable, you can optionally also give the variable an initial value. You can see that we've done this with the `slope` and `yint` variables.

The word `double` means that these variables will hold "double-precision floating-point numbers". Don't worry too much about what that means right now. It's enough to know that these variables will hold numbers with decimal points in them. Programmers call numbers that contain decimal places "floating-point numbers." [13]

Once you've defined a variable, you can use it in your program. For example, you can assign a value to it using an equals sign, as in "`x = 4.3`". This statement means "set the value of x equal to 4.3". The statement "`y = slope * x + yint`" does the math on the right-hand side of the equation and then sets the variable `y` equal to the result.

Later on, we'll learn how to ask the user for numbers, so we'll be able to ask the user to enter a value for `x`, instead of having the value written explicitly into the program.

We can use our new variables wherever we previously used numbers. Going back to the "`%lf`" format specifier in our printf statements, I'll now tell you that "`%lf`" means "insert a 'double' number here". The letters "`lf`" stand for "long float", which is another way of saying "double-precision floating-point number".

Finally, notice that we've defined our variables near the top of our program. Variables must be defined before you can use them, and some C compilers require that you define *all* variables before you do anything else in the program. Going back to our recipe analogy, you might think of these variable definitions as the list of ingredients. After we've listed the ingredients, then we can get down to the business of describing how to combine them into a tasty dish.

Figure 1.12: "La Tailleuse de Soupe", François Barraud (1933).

Source: Wikimedia Commons

[12] Variables are stored in the computer's *memory*, which is a temporary storage area that's erased whenever you restart the computer. This is unlike *files*, which are permanently stored on the computer's hard drive.

[13] In this book we'll only use three or four types of variables, although there are a lot more than that available.

1.8. A Note About Algebra

Let's pause for a minute and look at the way math is done in C programs. In the example above, we wrote "`y = slope * x + yint`". This looks an awful lot like equations we've seen in algebra.

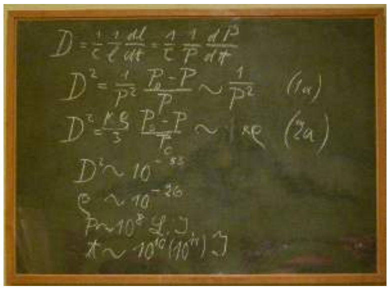


Figure 1.13: A blackboard used by Albert Einstein.

Source: Wikimedia Commons

One obvious difference is that we tend to use longer variable names in C programs than in algebra. When we're doing algebra, we usually write equations by hand, either on paper or on a blackboard, and we save time and effort by using single-letter symbols for variables whenever possible.

When typing a computer program, it doesn't take much effort to use longer, more descriptive names for our variables. This can help prevent us from getting confused as we're writing the program, and it makes it easier for other people (or our future selves) to look at the program and understand it.

A second, less obvious difference involves the actual meaning of an expression like "`y = slope * x + yint`". In algebra, this expression would mean something like "I promise you that the value of `y` is equal to `slope * x + yint`." On the other hand, in a C program, this expression means "I *command* you to *make* `y` equal to `slope * x + yint`."

The difference becomes apparent when you encounter an statement like "`x = x + 1`" in a C program. This statement would make no sense in algebra. There's no value of `x` for which `x = x + 1`. But in C, it makes perfect sense: We're commanding the computer to give the variable `x` the new value `x + 1`. If `x` is equal to 3 before this statement, it should be equal to 4 after the statement.

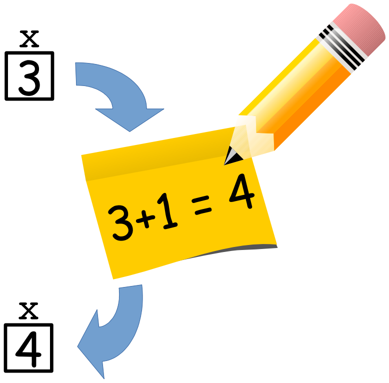


Figure 1.14: How the computer interprets the statement "`x = x + 1`". Remember that a variable in a C program is just a named storage location in the computer's memory. In this example, there's a variable named `x` that initially contains the value "3".

If we could look inside a computer's brain as it acts on the statement "`x = x + 1`" we'd see that it first calculates `x + 1`, saving the result in a temporary location, then copies the result into the variable `x`.

In later chapters you'll find that it's very important to remember that the equal sign in statements like this means *make* the left-hand side equal to the right-hand side.

In algebra the statements "$y = 2x + 3$" and "$2x + 3 = y$" are equivalent, but not in C. Remember that a C program is like a recipe: it's a set of instructions that should be followed in a particular order. "Pour milk into a bowl" isn't the same as "pour bowl into a milk"! The latter doesn't make any sense, just as the statement "$2x + 3 = y$" wouldn't make sense in a C program. In C programs, the equals sign *has a direction.* It takes whatever is on the right-hand side, and puts it into whatever is on the left-hand side.

1.9. Using Loops

We could use the program above to tell us the value of `y` at one particular value of `x`, but what if we want to look at how `y` varies as we change `x`? It would be nice if our program could print out, say, ten different `x` values and the corresponding `y` values.

We could, of course, do something like this:

```
x = 1.0;
y = slope * x + yint;
printf ( "At x = %lf the value of y is %lf\n", x, y );

x = 2.0;
y = slope * x + yint;
printf ( "At x = %lf the value of y is %lf\n", x, y );

x = 3.0;
y = slope * x + yint;
printf ( "At x = %lf the value of y is %lf\n", x, y );
```

et cetera, but it would be really tedious to type all of this. It would also be hard to change it later if we wanted a different set of `x` values, or if we wanted to use a different function for `y`.

Fortunately, if there's one thing computers are good at, it's doing the same thing over and over. That's why computers were invented. The C programming language lets us tell the computer to repeat a task a given number of times, optionally making small changes each time.

One way to do this in C is by using a "`for`" statement. Take a look at Program 1.2, named `loop.cpp`.

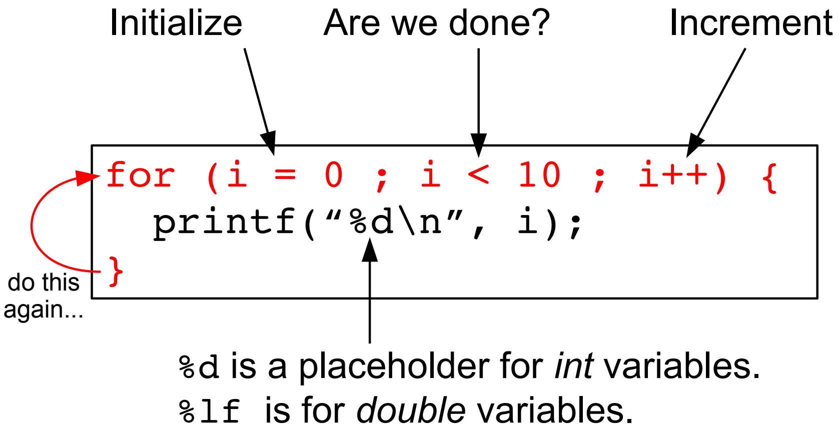
Program 1.2: loop.cpp

```
#include <stdio.h>
int main () {
      int i;

      for ( i=0; i<10; i++ ) {
              printf ( "%d\n", i );
      }
 }
```

First notice that we've defined a variable named "`i`". Instead of being a `double`, like the variables we've used before, this new variable is an `int`. That's short for "integer", which in the C language means the variable can hold numbers without decimal places.[14] Integers are the numbers we use to count discrete things, like apples or cars. They're the counting numbers, like 1, 2, 3,... including zero and negative numbers like -1. We're going to use the new variable to count how many times we've repeated a part of our program.

Programmers call a repeated part of a program a "loop". The computer starts at the "top" of the loop, does a list of tasks that are included in the loop, then goes back to the top of the loop and (optionally) starts again.[15] In principle, the computer could keep going around and around the loop forever, but we'll usually want to tell it to stop after it's gone around some number of times, or after some other requirement is met.

You can create a loop in your program by using a "`for`" statement. Figure 1.15 shows the anatomy of a `for` statement:



```
for (i = 0 ; i < 10 ; i++) {
  printf(“%d\n”, i);
}
```

`%d` is a placeholder for *int* variables.
`%lf` is for *double* variables.

Figure 1.15: The anatomy of a "`for`" loop. The first line marks the top of the loop. The bottom line marks the end of the loop. Everything in between is done repeatedly, some number of times.

In the first line, inside the parentheses after the word "`for`", we tell the computer three things that control how it will travel through this loop (see Figure 1.16). These are:

1. How to set things up before we start looping.
2. When to stop looping.
3. What changes to make each time we come to the bottom of the loop.

[14] You'll usually use `double` or `int` for numbers in your programs. Use `double` for any numbers that might have a decimal point, and `int` for integers.

[15] See the lyrics to "Helter Skelter" by the Beatles.

In Program 1.2, when we say "(i=0; i<10; i++)" we mean:

1. Before you start looping, set i equal to zero.
2. Keep going around the loop as long as i is less than ten.
3. Whenever you get to the bottom of the loop, add 1 to the value of i.

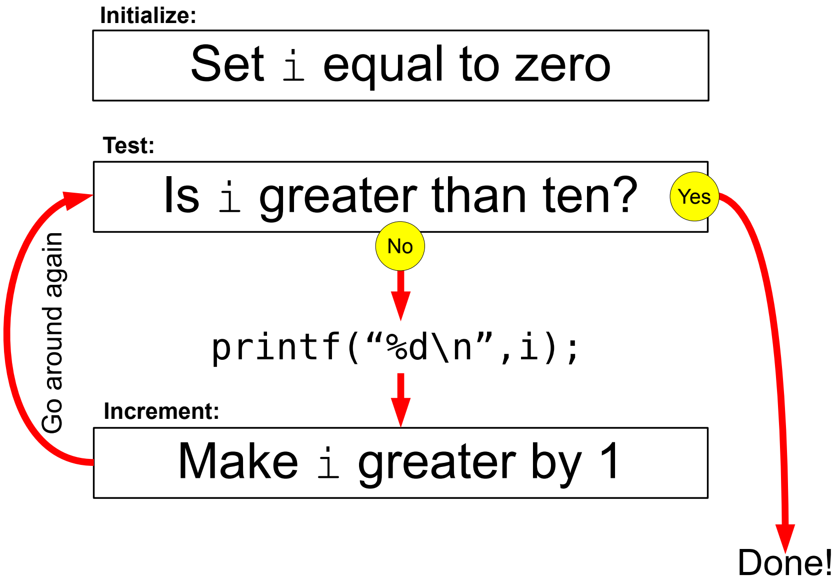


Figure 1.16: This diagram shows how a "for" loop works. Notice that if we gave i a value like 100 in the beginning, the program would never do the printf. Instead, it would just skip the loop entirely. This is important, because later on we'll encounter another kind of loop that will always be acted on at least once.

The mysterious-looking statement "i++" means "set i equal to i + 1". In C, "++" is the *increment operator*. (There's also a *decrement operator*, "−−", that decreases a variable's value.) The expression "i++" is just a handy shortcut here. It's exactly equivalent to saying "i = i + 1".

In the example program, we just print out the value of i each time we go around the loop. Notice that, instead of "%lf" in the printf statement, we use "%d". The "d" stands for "decimal integer", and it's what printf uses as a placeholder for an integer value. int variables go with "%d", and double variables go with "%lf". These are the only kinds of numerical values we'll use for most of the exercises in this book.

Exercise 4: Using Loops

As you did before with hello.cpp, create Program 1.2 by typing it into *nano*. When you're done typing, press ^X to exit *nano*. When asked what to call the new program, say "loop.cpp". Then compile your new program by typing:

```
g++ -Wall -o loop loop.cpp
```

If you see any errors, use *nano* to correct them, and try compiling again. When you've successfully compiled the program, run it by typing "./loop". What do you see? The program should print out a list of numbers, from zero to nine.

But what about...?

One more thing you should notice about Program 1.2: Look at the way we've indented the lines. This isn't necessary, but it's a good idea to keep your code neat and readable. Indenting the lines inside a loop can help you see where the loop begins and ends. When you write more complicated programs, you'll find that this often makes it easier to catch mistakes.

Pay attention to the way all of the examples in this book are formatted. Even if you don't use the same "programming style", you'll find it very useful to have a consistent style of some kind when writing your programs.

1.10. Calculations Inside a Loop

Now let's apply our knowledge of loops to the problem of finding the value of y for several values of x. Program 1.3 shows one way to do it.

Program 1.3: line.cpp

```
#include <stdio.h>
int main () {

      double x;
      double y;
      double slope = 2.0;
      double yint  = 3.0;

      int i;

      x = 0.0;

      for ( i=0; i<10; i++ ) {
              y = slope * x + yint;
              printf ( "%lf %lf\n", x, y );
              x = x + 1.0;
      }

 }
```

Before we start this program's "for" loop, we set the value of x to be zero. Then, each time we go around the loop we calculate the value of y, using "slope", "yint" and "x", and we add 1.0 to the value of x. The next time around, we use the new x value to calculate a new y value. After we've done this ten times, we stop.

Exercise 5: Doing Math Inside a Loop

As you've done before with the programs hello.cpp and loop.cpp, create the new program line.cpp using *nano* and compile it by typing "g++ -Wall -o line line.cpp". (If you see any errors, use *nano* to correct them, and try compiling again.) Run the program by typing "./line". Do you see what you expect? The program should print out a list of X and Y values.

But what about. . . ?

Notice that we change the value of x by saying “`x = x + 1.0`”. Could we have used C’s increment operator to do ths same thing, by just saying “`x++`” on this line? In principle, yes, that would work fine, but many programmers prefer not to use “++” with numbers that have decimal places (“floating-point” numbers, as programmers call them). As we’ll see later, we sometimes need to keep in mind the limits of the computer’s abilities. A computer can’t store all of the infinitely-many decimal places that a real number actually has. Instead, the computer needs to truncate the number to some manageable length. For example, instead of

```
3.14159265358979323846264338327950288419716939
9375105820974944592307816406286208998628034825
3421170679821480865132823066470938446095505822
3172535940812848111745028410270193852110555964
4622948954930381964428810975665933446128475648
2337867831652712019091... et cetera
```

the computer might approximate the number as 3.14159265358979. Because of this limitation on the precision of real numbers, small errors are introduced into the calculations done by the computer. A result that should be (by our knowledge of arithmetic) equal to 1.0000000..... will turn out to be (as seen by the computer) 1.0000000000001 or 0.999999999999. This kind of thing makes computer programmers cautious when incrementing, decrementing or (especially) comparing floating-point numbers. Avoiding the use of “++” with floating-point numbers helps us keep in mind that they aren’t the same as counting numbers, where the computer always has a well-defined, exact, “next number” to go to.

1.11. Graphing Our Results

Program 1.3 should print out a list of X and Y values, but how do we know they're the right ones? How do we know that our program is doing the right thing? The formula for calculating the Y values was `y = slope * x + yint`, which is the equation of a straight line. One way to check our program's output would be to see if the X,Y values it generates fall on a straight line.

Exercise 6: Making Graphs

To do this, we can use a third command-line utility (in addition to *nano* and *g++*, which we've already used) and a particular command-line trick. The command-line trick is this: Instead of just typing "`./line`" to run your program, type:

```
./line > line.dat
```

You won't see anything printed on your screen. Instead, the things that the program would otherwise have printed will be saved in a new file named `line.dat`.

The new command-line utility we'll use is *gnuplot*, which will let us make graphs of data. To start it, just type "*gnuplot*". You'll see something like this:

```
G N U P L O T
Version 4.2 patchlevel 6

gnuplot>
```

The "`gnuplot>`" at the bottom means that *gnuplot* is waiting for us to give it a command. Now type:

```
plot "line.dat"
```

This should show you a nice straight line of points, more or less like the picture in Figure 1.17.

If we'd like to draw a line through the points, we could type:

```
plot "line.dat" with linespoints
```

("`with linespoints`" means draw a symbol at each point, and draw a line connecting them.)

When we're done with *gnuplot*, we can leave it by typing "`quit`".

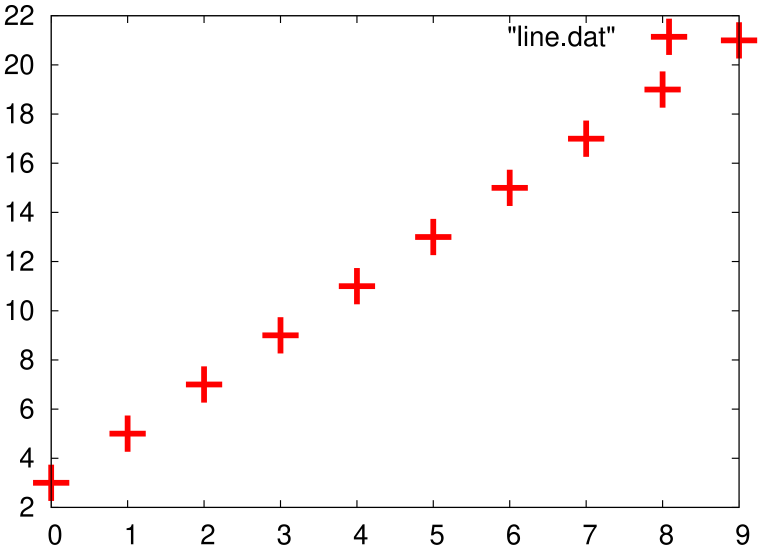


Figure 1.17: The result of typing `plot "line.dat"` in *gnuplot*.

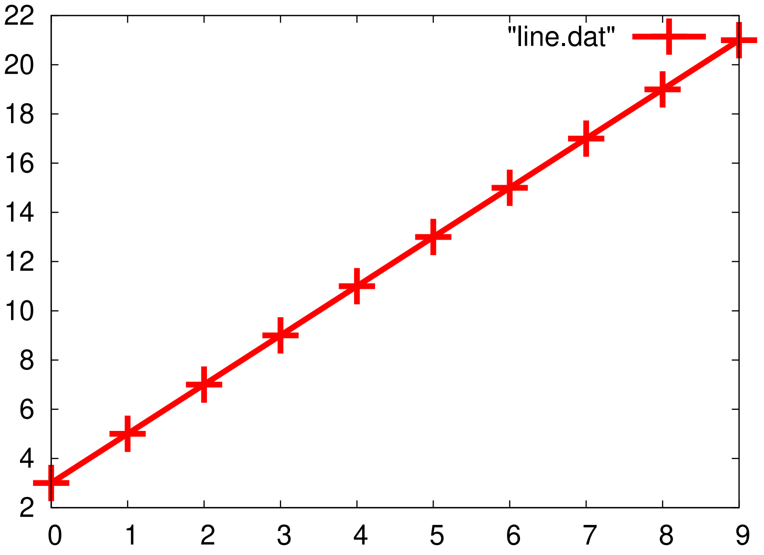


Figure 1.18: The result of typing `plot "line.dat" with linespoints` in *gnuplot*.

1.12. More About Variables

To understand how your programs use variables, you need to know a little about the computer's memory.

In computer terminology, *memory* is a temporary storage area that programs can use. It's a kind of scratch pad on which the program can scribble some information that it will need while it's working. The computer's memory consists of may *bits* that be turned on or off. (Think of a long, long line of thousands of light switches.)

When you use a variable in a program, the computer reserves some of those bits for storing whatever value you want to assign to that variable (for example, the number "11.6"). How many bits are reserved, and how they're used, depends on the type of variable.

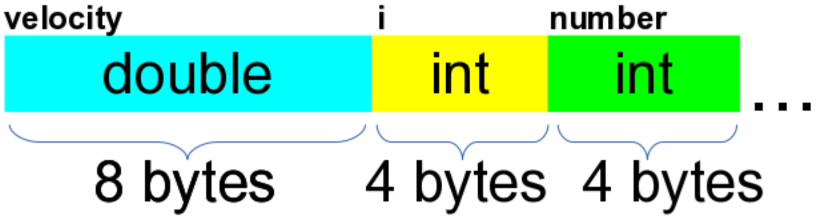


Figure 1.19: How a computer might store three variables in memory.

Figure 1.19 shows how the storage space for variables might be arranged if you wrote a program with a `double` variable named "`velocity`", and two `int` variables named "`i`", and "`number`". (Remember that a *byte* is just a group of eight bits.) Different types of variables are given different amounts of space. Bad things can happen if you try to put the wrong type of data into a variable.

For example, what would happen if you tried to stick a `double` value into the variable named "`i`", above? If you succeeded, the data would spill over into the adjoining variable ("`number`") and corrupt it.

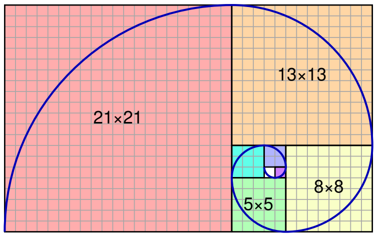
The C compiler tries to prevent this sort of thing two ways:

- It warns you when try to stick the wrong type of data into a variable, and
- It tries, when reasonable, to re-cast your data into a format that's appropriate for the variable into which you're putting it.

This re-casting can sometimes cause unexpected effects. For example, if you try to set an *integer* variable equal to "3.1415", the computer might just automatically drop the decimal part and set the variable equal to "3". We'll look at this in more detail later.

1.13. Fibonacci Numbers

Let's use our new-found loopy powers to do a little more math. The Fibonacci numbers are the sequence $0, 1, 1, 2, 3, 5, 8, 13, ...$, where each term in the sequence is the sum of the preceding two terms. This sequence pops up in all sorts of unlikely places in mathematics. It's named for the 13^{th} Century mathematician Leonardo of Pisa (later nicknamed "Fibonacci"), who used the sequence in describing the month-by-month growth of a population of rabbits.



A spiral made from squares whose sides are Fibonacci numbers.

Source: *Wikimedia Commons*

We might write a program to print the first few numbers of the sequence like this:

Program 1.4: fib.cpp

```
#include <stdio.h>
int main () {
  int a = 0;
  int b = 1;
  int c;

  int i;

  printf( "%d\n", a );
  printf( "%d\n", b );

  for ( i=0; i<10; i++ ) {
    c = a + b;
    printf( "%d\n", c );

    a = b;
    b = c;
  }

}
```

These variables will hold three successive terms of the sequence at a time. We'll start with the numbers 0 and 1.

Print the first two numbers.

The next number is the sum of the preceding two numbers.

b and c become the new first and second numbers, then we just keep repeating this process.

The program progresses by keeping track of three numbers at a time, in the variables named a, b, and c. It starts with 0 and 1 in a and b, respectively, then calculates the next number, c, by adding them. After printing the value of c the program "shifts" the numbers by one space, giving a the value of b, and b the value of c. Then it goes around the loop again, and comes up with a new value for c, the next number in our sequence.

If you compile program 1.4 and run it, it should print the first ten

Fibonacci numbers, like this:

```
0
1
1
2
3
5
8
13
21
34
55
89
```

A statue of Leonardo of Pisa, also known as "Fibonacci".

Source: Wikimedia Commons

Great! Since that went so well, what would happen if we tried to print more terms in this sequence? We could modify the "for" statement to make it do 100 terms instead of ten:

```
for ( i=0; i<100; i++ ) {
```

If we compiled this new version of the program and ran it, we'd see that things start off fine, but about halfway through something goes wrong:

```
...
165580141
267914296
433494437
701408733
1134903170
1836311903
-1323752223
512559680
-811192543
-298632863
...
```

What's going on here? If you refer back to Figure 1.19 in the preceding section, you might find a clue. Computers can't store infinitely big numbers. Each kind of variable has only a limited amount of space in the computer's memory. If the value keeps getting bigger and bigger, eventually it will be too big for the computer to store in that variable, and strange things will happen. But don't despair! The "int" and "double" variables we'll be using for most of our programs will be plenty big enough to hold the numbers we need, and later in the book, in Chapter 13, we'll see some techniques for storing humongous numbers.

Practice Problems

1. Write a program like Program 1.1 (`hello.cpp`), but instead of "Hello World!" make your program print your name. Call the program `myname.cpp`.

2. Write a program like Program 1.1 (`hello.cpp`), but instead of writing "Hello World!" make your program print the following address:

```
Mr. Sherlock Holmes
Consulting Detective
221b Baker St.
London NW1 6XE
```

   The address should appear exactly as it's written above. Remember that you can use "`\n`" to move to the beginning of a new line. Call your program `sherlock.cpp`.

3. Write a program that has a `double` variable named `age`. Give the variable a value equal to your current age, in years. Have the program write out the text "When I am twice my current age I will be ... years old", where "..." is replaced by twice your current age, as calculated by the computer. Call the program `myage.cpp`.
   Hint: Remember that `printf` uses `%lf` as a placeholder for `double` values, as shown in Section 1.6.

4. Repeat the previous problem, but this time have the program write out the text "When I was half my current age I was ... years old", where "..." is replaced by half your current age, as calculated by the computer. Call the program `halfage.cpp`. (Note that the symbol for division in C is "/".)
   Hint: Remember that `printf` uses `%lf` as a placeholder for `double` values, as shown in Section 1.6.

5. Using Program 1.2 (`loop.cpp`) as a model, write a program that prints out the words "I'm a programmer!" ten times, with each sentence on a separate line. Call the new program `cheers.cpp`. (Check to make sure your program prints the text the correct number of times.)

6. Write a program named `printvals.cpp` that sets the values of three integer variables and then prints those values, using `printf` with a `%d` placeholder. The variables should be named `x`, `y`, and `z`, and they should be given the values 1, 2, and 3, respectively. When the program is run, it should print exactly this output:

```
x is 1
y is 2
z is 3
```

Here's a picture of a "computer". That was Katherine Johnson's title when she worked for NASA. She was one of many mathematicians who did, by hand, the tedious calculations required to successfully navigate spacecraft into orbit and back to earth. She worked on the Apollo 11 mission to the moon, and her calculations helped bring the aborted Apollo 13 mission safely back to earth. Even after electronic computers came into use, human computers like Katherine Johnson were asked to check the results that came out of their electronic counterparts.

Source: Wikimedia Commons

The Sherlock Holmes Museum at 221b Baker Street.

Source: Wikimedia Commons

7. Write a program named `ndots.cpp` that has an integer variable named `n`. Have the program use a `for` loop to print as many dots as the value of `n`, in exactly this format:

   ```
   Here are 10 dots:
   ..........
   ```

   Note that the output should start, as shown above, by saying how many dots the program is going to print. Also note that after printing all of the dots, the program's output should go to the next line. Make sure that, if you change the value of `n` and recompile the program, the program's output changes to match the new value of `n`.

8. Using Program 1.2 (`loop.cpp`) as a starting point, write a program called `countdown.cpp`. Change **just the `printf` line** to make the new program print the following:

   ```
   10...9...8...7...6...5...4...3...2...1...
   ```

   Hint 1: Remember that you can use an arithmetic expression in a `printf` statement, as shown in Section 1.6.
   Hint 2: Remember that you can add or remove `\n` in a `printf` statement to control whether it goes to the next line after printing some text, as shown in Section 1.5.

9. What if we wanted Program 1.2 (`loop.cpp`) to start at 100 and count to 1000 by hundreds (100,200,300,... up to 1000)? How could we do that without changing the "`for`" line in this program? Write a new program with these changes, and call it `loop2.cpp`.

10. Using Program 1.2 (`loop.cpp`) as a model, write a program that prints out a list of all the numbers from zero to 999 and the cube of each of these numbers. The format of the output should be lines like this:

    ```
    0 0
    1 1
    2 8
    3 27
    4 64
    ...
    ```

    where the second number in each line is the cube of the first number. Hint: One way to cube a number in C is simply to multiply it by itself twice, like this: `2*2*2`. Call your program `cubes.cpp`.

    You can use *gnuplot* to check the program's results. First, send the program's output into a file, like this:

    ```
    ./cubes > cubes.dat
    ```

    Then start *gnuplot* and give it the command:

    ```
    plot "cubes.dat" with lines
    ```

    The result should look like Figure 1.20.

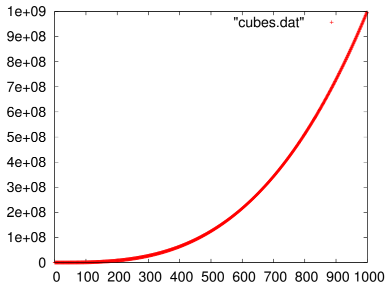


Figure 1.20: The output of your `cubes` program plotted by *gnuplot*.

11. Using Program 1.3 (`line.cpp`) as an example, write a program named `curve.cpp` that prints values of x between -50 and 50 (in increments of 1), along with the value of $y = 200 + x^2/3$ for each x value. (Note that the symbol for division in C is "/".)

    Note that you won't need the variables `slope` and `yint` from Program 1.3. You'll also need a slightly different `for` statement, since this loop will cover 100 values instead of only 10. You might find it useful to know that one way to square a number in C is simply to multiply it by itself, like "`x*x`".

    The program should print the x and y values in two columns, like this:

```
-50.000000 1033.333333
-49.000000 1000.333333
-48.000000 968.000000
-47.000000 936.333333
-46.000000 905.333333
...
```

    You can use *gnuplot* to check the program's results. First, send the program's output into a file, like this:

```
./curve > curve.dat
```

    Then start *gnuplot* and give it the command:

```
plot "curve.dat" with lines
```

    The result should look like Figure 1.21.

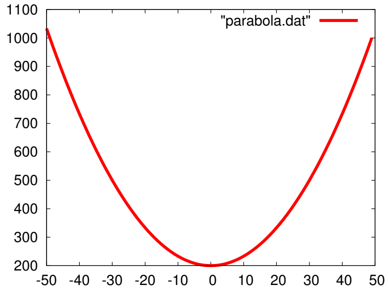


Figure 1.21: The output of your `curve` program plotted by *gnuplot*.

12. Make a new program named `pell.cpp`. Start by copying Program 1.4 on Page 46. Then modify the program so that it:

    (a) Starts with `a = 2` and `b = 6`, and

    (b) Instead of adding the preceding two numbers, as Program 1.4 does, add the first number to *twice* the second number.

    When you compile and run your program it should print a sequence of numbers like 2, 6, 14, 34, 82, ... These are the "companion Pell numbers[16]". They're related to the Fibonacci numbers, and can be used to find approximate values of the square root of 2.

13. Write a program named `2power.cpp` that prints the values of powers of 2, from 2^0 to 2^{15}. To do this, use two integer variables named `i` and `x`. Start by setting the value of `x` to 1. Then use a `for` loop that sets `i` to each of the values from zero to 15. For each value of `i`, print `i` and `x`, then multiply `x` by 2. The program's output should look exactly like Figure 1.22.

```
0 1
1 2
2 4
3 8
4 16
5 32
6 64
7 128
8 256
9 512
10 1024
11 2048
12 4096
13 8192
14 16384
15 32768
```

Figure 1.22: The output of the `2power` program.

[16] See this Wikipedia article.

Writing Pretty Programs

The C programming language gives you a lot of freedom in how you write your programs. There aren't any rules about how lines should be indented, for example, and you can choose to write long statements as one long line or break them up over multiple lines. Two programs that do exactly the same thing can look very different. For instance, here's another way we could have written Program 1.3 (`line.cpp`):

```
#include <stdio.h>
int main () { double x; double y; double slope = 2.0;
double yint  = 3.0; int i; x = 0.0;
for ( i=0; i<10; i++ ) {y = slope * x + yint;
printf ( "%lf %lf\n", x, y ); x = x + 1.0;}}
```

I think you'll agree that this is harder to read than the earlier version. Here are four rules for writing pretty programs:

The Cheyenne or Arapaho woman named Pretty Nose was a war chief who fought at the Battle of Little Bighorn. Her grandson, Mark Soldier Wolf, was a U.S. Marine who fought in Korea. Pretty Nose was 101 years old when he returned home, and greeted him with a war song.

Source: Wikimedia Commons

Rule 1: Use Indentation

For making your programs pretty, the most important thing you should remember is that programs are made out of parts that *can hold other parts inside them*. When writing a program we use indentation to make it clear that some parts are inside of others[17]

In a C program, curly brackets tell the computer that something is contained inside something else. For example, Program 1.3 consists of a main program that has a `for` loop nested inside it. The statements inside the `for` loop are almost like a little program that the main program runs ten times:

`#include <stdio.h>` { Main Program { A `for` Loop } The Rest of the Main Program }

[17] You'll find lots of other tips for writing pretty programs here: *https://www2.cs.arizona.edu/ mccann/indent_c.html*.

As you're writing your programs, when you see an opening bracket, {, start indenting. When you see a closing bracket, }, stop indenting. If you're in an already-indented section and you see another opening curly bracket, add more indentation:

Curly brackets are also called "braces", a word that originally meant "arms", as in "embrace". The curly brackets in our programs embrace the statements they enclose.
(Louise Élisabeth Vigée Le Brun, Self-portrait with Her Daughter, Julie, c. 1789).
Source: Wikimedia Commons

```
#include <stdio.h>
int main () {

     double x;
     double y;
     double slope = 2.0;
     double yint  = 3.0;

     int i;

     x = 0.0;

     for ( i=0; i<10; i++ ) {
            y = slope * x + yint;
            printf ( "%lf %lf\n", x, y );
            x = x + 1.0;
     }

}
```

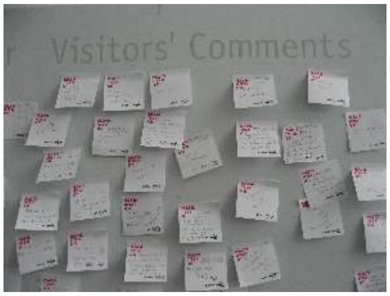
It doesn't matter how much space you use for indentation. Some people like to use a tab for each level of indentation. Other people prefer something "shallower", maybe only a couple of spaces. Either way is OK. Just be consistent inside each program you write.

Rule 2: Collect Variable Definitions

At this stage in your programming career, I recommend that you define all variables at the top of your programs, right under the "`int main ()`" statement. Later on you'll learn that C++ allows you to define variables anywhere in a program, and there are advantages to using that ability, but pure C compilers don't allow this. If you want your programs to be as portable as possible, stick to defining variables at the top for now. This also gives you one handy place to look to see your variable definitions.

Rule 3: Use Comments

“Comments” are text that you put into your program to explain what the program does, tell people who wrote the program, give advice about how to run the program, warn about copyrights and patents, or anything else you want to say. Comments are just ignored by the compiler, so they don’t affect the way your program runs. As far as the compiler is concerned, the comment isn’t even there.



Source: Wikimedia Commons

The *g++* compiler lets you add comments to your program in a couple of ways. Here’s one of them:

```
#include <stdio.h>
int main() {
        // This program was written by Bryan Wright.
        int i;
        for (i = 0 ; i < 10 ; i++) { // Start loop.
                printf("loop number %d\n", i);
        }
}
```

Almost any text between a double slash (//) and the end of the line is a comment. I say “almost” only because this doesn’t work inside quotes, so that:

```
printf("Hello World! // and some other stuff");
```

would print out “`Hello World! //and some other stuff`”.

The second way to add comments is an older one that will work in any compiler that understands C or C++, and it has the advantage that comments can extend over multiple lines. Look at the following example:

```
#include <stdio.h>
int main() {
        int i;
        /* This program was written by Bryan Wright.
           Copyright 2015.
           All rights reserved.
           Seriously.  I'll call my lawyer.
           Don't mess with me. */
        for (i = 0 ; i < 10 ; i++) {
                printf("loop number %d\n", i);
        }
}
```

Any text between **/*** and ***/** is a comment. One caveat is that you can't have a comment inside another comment, so something like:

```
/*Some words /* and some more words */and the end */
```

would cause the compiler to complain, and refuse to compile your program.

Comments are a great way to make your program more readable, but they're also very useful for temporarily removing parts of your program. If the program contains a line that we want to keep, but temporarily disable, we can just put a "//" at the beginning of the line. You'll find that this can help you find problems in your programs. If something isn't working right, you can selectively turn off parts of the program to help you find the problem.

Rule 4: Look Around You[18]

It's possible that someday you'll join a research group or a business where other people have already written lots of programs. When you start contributing your own programs, it's important that your programming style matches the stylistic conventions that are already in use. If everybody uses tabs for indentation, you should probably do so, too. This is especially important if you start modifying programs that other people have written.

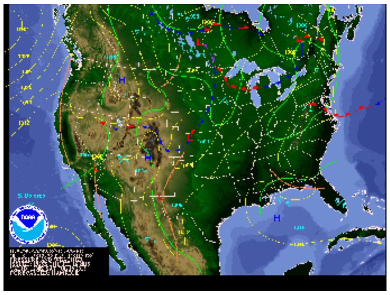
Figure 1.23: Now you're programming with *style!*
(Fred Astaire and his sister Adele cutting a rug in 1921.)
Source: Wikimedia Commons

[18] Not to be confused with the BBC series of the same name, which you should watch if at all possible: *https://www.youtube.com/watch?v=gaI6kBVyuoo*

2. *Random Numbers and Simulations*

2.1. Introduction

Some of the world's most powerful computers and most sophisticated software exist for the purpose of telling you whether you need to carry an umbrella tomorrow. Weather predictions demand extreme computing power. These predictions are made by simulating the earth's atmosphere. They begin with current weather conditions (temperature, pressure, humidity, wind speed) at many locations around the world and at different heights within the atmosphere. Then they approximate the atmosphere by pretending it's made of millions of discrete "cells", and the behavior of each of these cells is simulated as it changes over time. Simulations like this allow us to find approximate answers to problems that would be difficult or impossible to solve exactly.



A weather forecast.

Source: NOAA

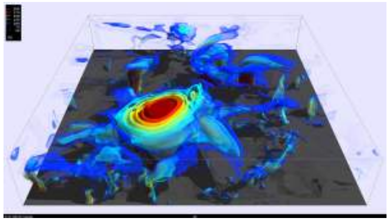
Computer simulations often make use of random numbers. If you've ever played a video game (or watched a movie with computer-generated special effects) you've seen images made with the help of random numbers. The trees in a video game forest probably aren't drawn by hand. They're generated from a recipe that uses random numbers to decide where to put the branches and leaves, how tall the tree is, and its location in the forest.

Computer-generated trees.

Source: Wikimedia Commons

Simulations can let us take random numbers, combine them with a few simple rules that describe how neighboring components interact with each other, and turn that into a prediction about the complex behavior of a large system.

In this chapter we'll learn how to create programs that use random numbers to simulate processes in the real world.



A computer simulation of twisted magnetic fields in the Sun's atmosphere.

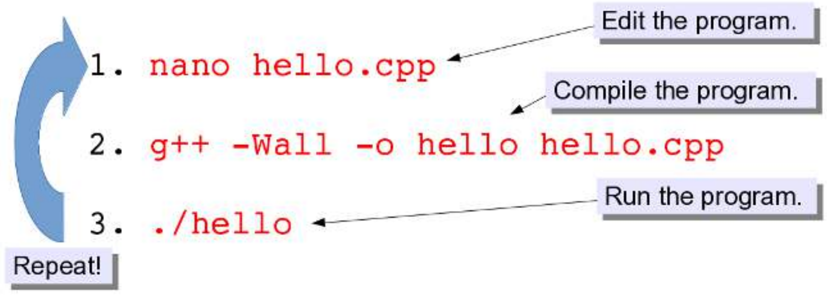
Source: Tim Sandstrom, NASA/Ames

2.2. The Code Development Dance

In the last chapter we saw how to create programs using an *editor* and a *compiler*. The process of creating a program is usually a loop, like the loops we created inside our programs. We start out by writing some statements in the C language and saving them into a file, then we compile the file and run the resulting binary version of the program. If the program doesn't do what we want it to do, we go back and edit some more, then try again until we have a working program. I call this process "The Code-Development Dance" (see Figure 2.1).

Dance in the Moonbeam by Theodor Kittelsen.

Source: *Wikimedia Commons*



```
1. nano hello.cpp
2. g++ -Wall -o hello hello.cpp
3. ./hello
```

Figure 2.1: The Code-Development Dance

No matter how far you go in programming, you'll still follow this same process while developing programs.

Programmers often refer to the instructions in a computer program as "code". The C language statements you've written are called "source code" and the binary files created by the compiler are called "binary code"

In the exercises that follow, we'll be working on two new programs. In each case, we'll start out with a simple version of the program, then make improvements. Each time we change something, we'll go through the process of editing our program, compiling it, and running it. Refer back to Figure 2.1 if you need help.

2.3. Using the `rand` Function

Take a look at Program 2.1, named `rand.cpp`. This program is similar to the loop programs we've written previously, but it introduces two new things. First, at the top of the program there's an extra `#include` statement. Second, the program makes use of a new function, called `rand`.

Program 2.1: rand.cpp (Version 1)

```
#include <stdio.h>
#include <stdlib.h>
int main () {
        int i;
        for ( i=0; i<10; i++ ) {
                printf ( "%d\n", rand() );
        }
}
```

Notice that `rand` is a function, like `printf`, but it's a function that takes no arguments. It just generates random numbers out of nothing.

Exercise 7: Random Numbers

Write and compile Program 2.1, using *nano* and *g*++, then run it to see what it does.

You should find that the program generates a list of seemingly random numbers. That's the whole purpose of the `rand` function. Each time your program uses `rand`, it gives you a different number.

Try running your program several times. Do you notice anything surprising?

Here's a useful tip: If you want to run your program again without having to type "./rand", you can use the up arrow key on the keyboard to bring back commands you've used before. Just keep pressing the up arrow until you see the command that you want to re-do, then press enter to repeat that command. You can also use the left and right arrow keys to move back and forth in what you've typed and make changes before you press enter.

Before we can use `rand`, we need to add the extra `#include` statement at the top of the program. This statement tells the C compiler some necessary information about the `rand` function. The first `#include` statement, which we've used in our earlier programs, provides the compiler with information it needs in order to use the `printf` function. We'll learn more about these `#include` statements in later chapters.

2.4. Making it Better

If you run Program 2.1 several times, you should find that, although the numbers look random, you get the same set of numbers each time you run the program. That doesn't seem very random, does it? Let's try to do better. Take a look at Program 2.2.

In Program 2.2 we've added two more lines. Before the "`for`" loop there's now a cryptic-looking statement involving two new functions, `srand` and `time`. Then, at the top, we've added yet another `#include` statement.

Program 2.2: rand.cpp (Version 2)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
        int i;
        srand(time(NULL));
        for ( i=0; i<10; i++ ) {
                printf ( "%d\n", rand() );
        }
}
```

Exercise 8: More Random!

What do these changes do? Let's try it. Remember that you can modify your program by typing "`nano rand.cpp`", then make your changes, and press Ctrl-X to save your changes and exit *nano*.

Edit your `rand.cpp` program, compile it again and then try running it several times. (Wait at least one second between tries.) You should now see that you get a different set of numbers each time you run the program. That's great, but how did it happen?

Note for Mac users: If you're using an Apple computer, you might find that Program 2.2 still gives you the same set of "random" numbers every time you run it. If you waited, say, an hour and ran it again, you'd find that *then* the program would give you different numbers. This happens because of a peculiarity in the way `rand` and `srand` work on the Mac. To fix the problem, change

```
srand(time(NULL));
```

to be

```
srand(time(NULL));
srand(rand());
```

2.5. Pseudo-Random Numbers

Let's think about what we mean by "random". If we roll a fair die, it should be impossible to predict which number will come up. Even if we roll the die many times, the outcome of the next roll should be unpredictable and independent of all the previous rolls. If the numbers are really random, it should be impossible to predict what the next number will be.

Rolling a fair six-sided die will give you a truly random number between 1 and 6, inclusive.

Source: *Wikimedia Commons*

It's not possible to generate truly random numbers using only a computer program. A function like `rand` can ultimately only do math, and we can expect that the same set of mathematical operations will always give the same answer. The `rand` function starts with an initial number (called a "seed") and then just does some very roundabout calculations that give us another number that has no *obvious* relation to the preceding number. Thereafter, each time we use `rand` in our program it builds on the number it had before.

Our first program gave us a chain of seemingly random numbers, but because the seed gets set to the same value each time we start the program, the list of numbers was always the same. The second version of the program sets the seed to a different value each time we run the program. It does this by using the computer's clock. Whenever we run Program 2.2 the seed is set to the current time, expressed as the number of seconds that have elapsed since January 1, 1970.[1] That's what "srand(time(NULL))" does. The srand function sets the seed used by rand. The expression "time(NULL)" gives us the time. The extra #include statement tells the compiler what it needs to know in order to use the time function.

Even with this change, it's important to know that if your program generates millions or billions of numbers, rand will eventually start repeating itself. (See Figure 2.2.)

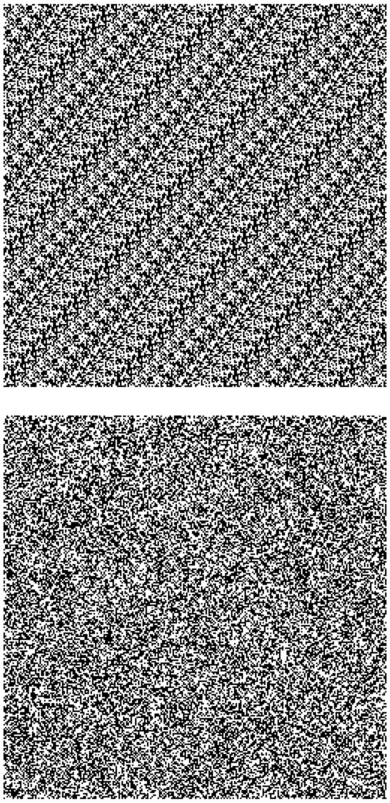


Figure 2.2: These two images show the output of a bad random number generator (top) and a better generator (bottom). The lines in the top image indicate that the generator soon starts repeating the same set of numbers. The generator used for the bottom image goes much longer without repeating.

Functions like rand are called "pseudo-random number generators" (PRNGs). The numbers they generate aren't really random, but they're good enough for many purposes. Some computers now include a device called a "true random number generator" (TRNG). These devices generate random numbers by observing real physical processes, such as thermal noise. They effectively roll real miniature dice to generate their random numbers. TRNGs are becoming more important because good random numbers are essential to cryptography.

2.6. Random Numbers Between Zero and One

You've probably noticed that the numbers generated by rand are large integers. That's fine for some things, but programmers often want to generate random real numbers that fall in the range between zero and one (for reasons that will soon become apparent). How can we do this using rand? Take a look at Program 2.3.

The rand function generates integers between zero and a large number called RAND_MAX. RAND_MAX is one of the things defined when we say #include < stdlib.h >.[2] If you're curious, you could print out the value of RAND_MAX with a statement like:

```
printf( "%d\n", RAND_MAX ).
```

Program 2.3 introduces a new variable, x. We'll want x to be a random number between zero and one, so this variable can't be an integer. Instead, we'll make it a double.[3] We calculate the value of x by

[1] This is why I told you to wait at least one second between tries. Otherwise you might run the program twice with the same seed, and get the same set of numbers.

[2] The numerical value of RAND_MAX may vary, depending on what version of the C compiler you use.

[3] See Chapter 1.

Program 2.3: rand.cpp (Version 3)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
        int i;
        double x;
        srand(time(NULL));
        for ( i=0; i<10; i++ ) {
                x = rand()/(1.0 + RAND_MAX);
                printf ( "%lf\n", x );
        }
}
```

getting a random integer from `rand` and dividing that number by `1.0 + RAND_MAX`. Since the numbers generated by `rand` are always between zero and `RAND_MAX`, `x` should always be between zero and something slightly less than one[4].

Note that it's important to say `1.0 + RAND_MAX` here instead of `1 + RAND_MAX`. To understand why, we have to think about the way C does arithmetic with integers. `RAND_MAX` and the numbers generated by the `rand` function are integers.

When C divides one integer by another, it assumes that you want the result to be an integer, too. If the result were equal to 0.7, the computer would drop everything after the decimal point and just leave zero. Since `RAND_MAX` is an integer, C would see the expression `1 + RAND_MAX` as an integer, and `rand()/(1 + RAND_MAX)` would always be zero. By just saying `1.0` instead of `1`, we give C a clue that we want to keep decimal places in our results.

Exercise 9: Making Real Numbers

Try modifying your program so that it looks like Program 2.3. Compile it, run it, and look at the results. You should now see a list of numbers that are all between 0 and 1.

[4] Why don't we want to go all the way to one? We'll see the benefits of that in a later chapter. For now, don't worry too much about it. Since `RAND_MAX` is a very large number, the biggest numbers we generate will be very close to one (less than a billionth smaller).

2.7. Random Integers Between Some Limits

Sometimes we want to generate a random integer between some minimum and maximum values. For example, maybe we want to simulate rolling a six-sided die, so we want to generate numbers between one and six.

We can do this by starting with a random real number between zero and one, as described in the preceding section. For example, we might have a `double` variable named `x` in our program, and a line that says:

```
x = rand()/(1.0 + RAND_MAX);
```

That would give `x` a random value between 0 and 0.9999999...[5]. We could multiply this by six to get a number between 0 and 5.9999999.... Let's create a new `double` variable named `y` that does that:

```
y = 6 * x;
```

C provides us with a way of chopping the decimal part off of a number. All we need to do is put `(int)` in front of the value. Let's modify our program so that we have an integer variable named `i` instead of the `double` variable named `y`:

```
i = (int)( 6 * x );
```

Notice that we've put parentheses around `6 * x` so that `(int)` applies to the whole thing. Otherwise, it would just apply to `6`. Before the `(int)` is applied, we have a random number between 0 and 5.9999999.... The `(int)` chops off the decimals and leaves us with a number between 0 and 5.

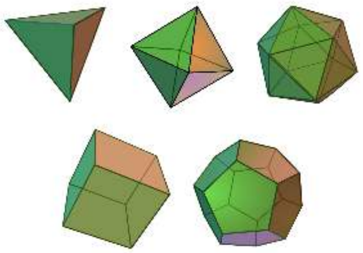
If our goal is to generate a number between 1 and 6, we just need to do one more thing: add 1 to the value of `i`.

```
i = 1 + (int)( 6 * x );
```

What if we wanted numbers between 2 and 7 instead of 1 and 6? Then we'd just need to change one thing:

```
i = 2 + (int)( 6 * x );
```

Notice that the multiplier, 6, didn't change. This is because the our new range still includes six possible values. Now they're 2, 3, 4, 5, 6, and 7. In general, if we want integers between n_{min} and n_{max}, the number of values will be $n_{max} - n_{min} + 1$.



Dice come in many shapes. Often they're shaped like one of the five platonic solids. These are the only regular convex polyhedra that are possible in three dimensions. In four dimensions there are six such shapes, but in five and higher dimensions, there are only three. See this excellent video by Carlo Sequin for some fun with higher-dimensional "polytopes": https://www.youtube.com/watch?v=2s4TqVAbfz4.
Source: Wikimedia Commons

[5] It never quite gets to 1.0 because the maximum value returned by `rand` is `RAND_MAX` and we're dividing by `1.0 + RAND_MAX`.

So, if we want to get a random integer between `min` and `max` we can do it like this:

```
nvals = max - min + 1;
i = min + (int)( nvals * x );
```

Program 2.4 uses this strategy to generate a random number between 1 and 6.

Program 2.4: diceroll.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
  double x;
  int i;
  int min = 1;
  int max = 6;
  int nvals;

  nvals = max - min + 1;

  srand(time(NULL));

  x = rand()/(1.0 + RAND_MAX);
  i = min + (int)(nvals*x );

  printf ( "%d\n", i );
}
```

This program could be modified to generate a random integer in any range you want, just by changing the values of `min` and `max`.

Exercise 10: Gonna Roll The Bones

Write a program based on Program 2.4 that rolls *two* six-sided dice and prints (1) the number on each die and (2) the sum of their two numbers. For example, if both dice roll six, the sum would be twelve. Run the program several times to see if you can roll a twelve!

Claus Meyer, 1886, *Die Würfelspieler*.

Source: *Wikimedia Commons*

2.8. Writing a Simulation Program

Imagine a rock in a gutter. In this place it rains once per day, and every time it rains the rock slides some random distance, Δx, down the gutter. Assume Δx is always between zero and 100 cm. Let's try to simulate this physical system with a computer program, and see how the rock behaves.

For more on stones in gutters, see the excellent short story "Fall of Pebble-Stones" by R.A. Lafferty.

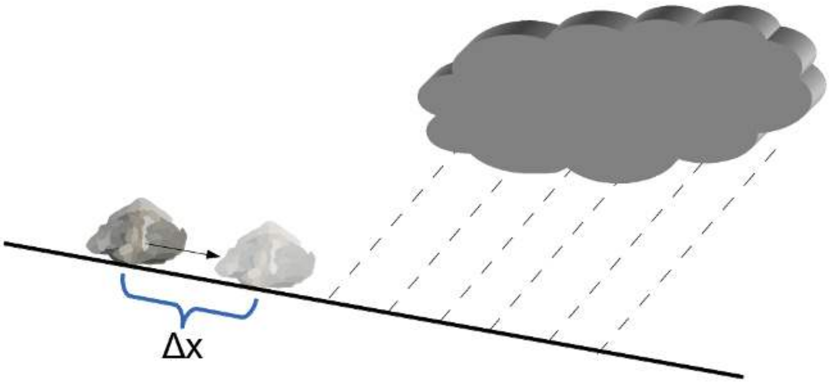


Figure 2.3: A rock, sliding along a gutter.

Program 2.5: gutter.cpp (Version 1)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
        int i;
        double delta_x;
        double x;
        srand(time(NULL));
        x = 0.0;
        for ( i=0; i<10; i++ ) {
                delta_x = 100.0 * rand()/(1.0 + RAND_MAX);
                x = x + delta_x;
                printf ( "%lf\n", x );
        }
}
```

You'll notice that Program 2.5 is very similar to Program 2.3. The main differences are that (1) we set the variable x equal to 0.0 before starting our loop, and (2) each time around the loop we add a random amount to x. Also, instead of x as our random number, we've renamed this variable delta_x.

As you saw in Chapter 1, when you see an expression like x = x+d in a C program it means "Set the new value of x equal to the old value plus d". Remember that this is a little different from what you may be used to in algebra. It might help if you keep in mind that, in C, the statement x = 1 means "assign the value 1 to the variable x". In algebra, on the other hand, the same statement would mean "I promise you that x is equal to 1".

The variable x stores the rock's current position, in centimeters. It starts out at `x = 0.0`. Each time around the loop represents one rainstorm, which washes the rock a random distance, Δx, down the gutter. We want Δx to be a number between zero and 100 centimeters, so we calculate it by taking a random number between zero and one (as we did in Program 2.3) and multiplying that by 100. The new value of x after the rainstorm is $x + \Delta x$. At the bottom of the loop we print out the new value of x. The program simulates the movement of a rock as it slides down the gutter over the course of ten days in this very rainy location.

Exercise 11: First Gutter Program

Try writing Program 2.5, compiling it, and running it. Do the values it prints out make sense? Run it several times (waiting at least one second between tries). You should get different, but still reasonable, results each time.

Each time you run it, the last number printed by the program is the stone's position at the end of day number ten. Do these numbers seem reasonable? Keep in mind that if the stone traveled exactly 50 cm each day (halfway between zero and 100 cm), it would end up 500 cm from the origin at the end of day ten.

But what about...?

In Program 2.5 we named one of the variables `delta_x`. What kinds of names are allowed for variables in the C language?

Allowed Characters:

Variable names can only contain letters (upper- or lower-case), numbers and the underscore character, "_". Names must begin with a letter or an underscore (not a number).

It's good practice to always use a letter as the first character in variable names. Leading or trailing underscores are sometimes used internally by the compiler. If you get into the habit of using an underscore at the beginning of variable names, you may run into confusion later in your programming career.

Remember that C is case-sensitive, so that a variable named `Velocity`, with an upper-case "`V`", is completeley different from a variable named `velocity`. Also, note in particular that spaces aren't allowed in variable names.

Maximum Length:

Different versions of the C compiler have different limits on the maximum length of variable names. The compiler we're using, *g++*, has no limit. In principle, you could give a variable a name that was thousands of letters long, although this would obviously be awkward to type! Some C compilers limit variable names to 2,048 characters, and others require that at least the first 31 characters of each name be different from any other name in your program. With all of that in mind, it would be a good idea to limit yourself to variable names that are 31 characters or fewer.

It's good practice to give your variables clear, concise names like `velocity`, `width`, `temperature`, *et cetera*. This helps you remember what they're for, and makes it easier for other people to understand your program.

Reserved Words:

Some names are simply not allowed. For one thing, you can't give your variable a name that's the same as any of the words that make up the C language. You couldn't, for example, name a variable `int`, `double` or `for`. There are 32 words of this type. For the record, they are:

> **auto break case char const continue default do double else enum extern float for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while**

You also can't give your variable the same name as any function your program knows about. It wouldn't be allowable to name a variable `printf`, for example, in any of the programs we've written so far.

(Note that I'm being careful to say "any function your program knows about". You'll understand what I mean later, when we talk about libraries of functions.)

2.9. Some New Arithmetic Operators

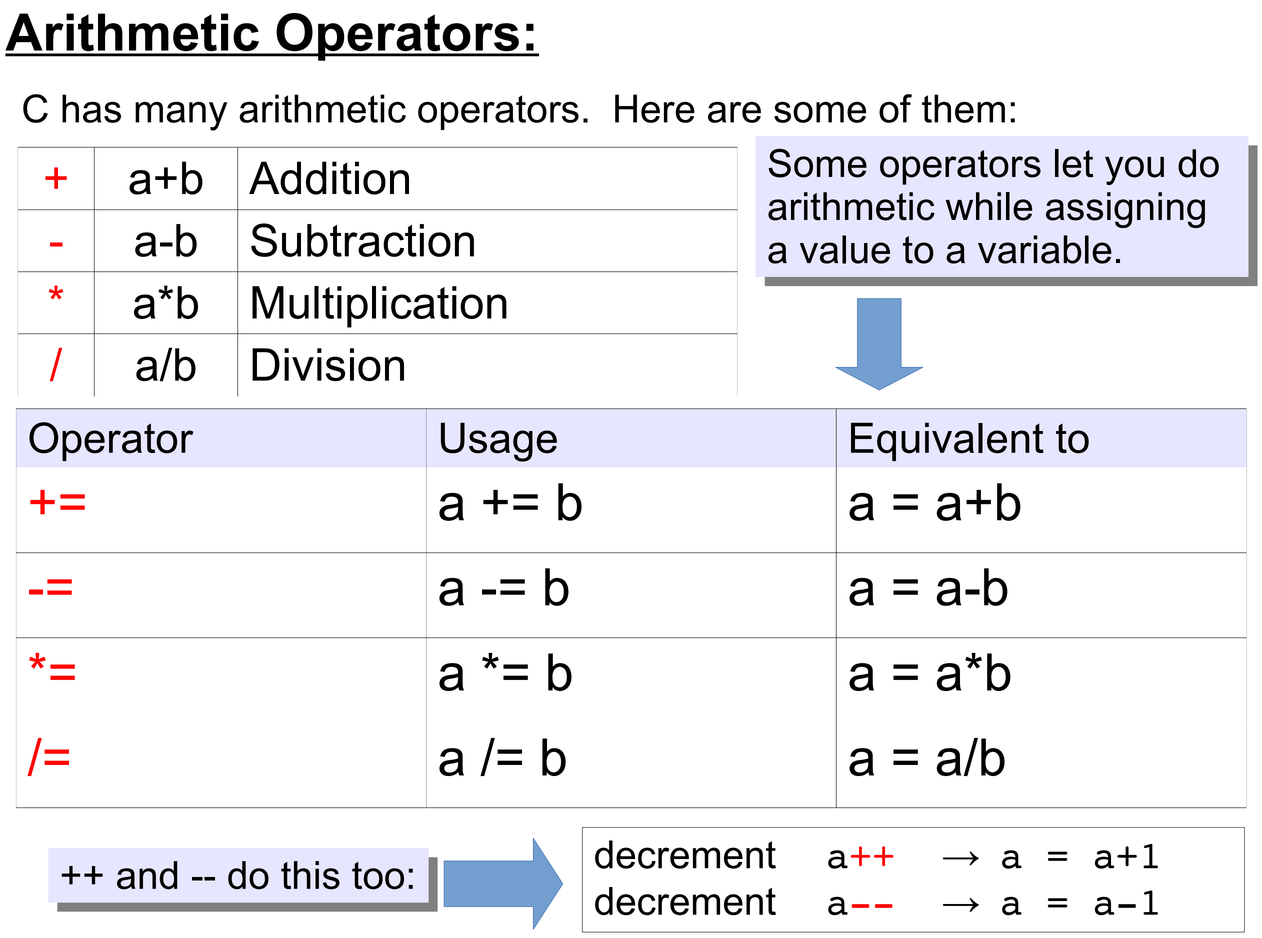
The C compiler understands many arithmetic operators. Besides +, −, ∗, and / there are several "combination" operators that provide shortcuts for doing common operations. Figure 2.4 shows some of these.

If we say, for example, `d += 100`, we mean "increment the value of `d` by 100". It's exactly equivalent to writing `d = d + 100`, but a little easier to type. I find that it also helps prevent typing errors, especially with long variable names. Consider the following for example:

```
somelongname = somelongnome + 10;
```

The += operator is similar to the ++ operator we've been using in "`for`" loops. The difference is that ++ increments the value by 1, but += can increment by any amount.

Did you catch the typo? If I'd written `somelongname += 10` instead, I'd have one less opportunity to misspell the variable name.



Arithmetic Operators:

C has many arithmetic operators. Here are some of them:

+	a+b	Addition
-	a-b	Subtraction
*	a*b	Multiplication
/	a/b	Division

Some operators let you do arithmetic while assigning a value to a variable.

Operator	Usage	Equivalent to
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b

++ and -- do this too:

```
decrement   a++   → a = a+1
decrement   a--   → a = a-1
```

Figure 2.4: Some of C's arithmetic operators.

2.10. Focusing on the Important Results

What if we're only interested in the total distance a stone has travelled at the end of ten days? We can modify our program, as shown in Program 2.6, so that instead of printing each new position, it only prints out the final position. As you can see, this just requires us to move the `printf` statement outside of the "`for`" loop.

Note that Program 2.6 also takes advantage of the += operator to make one of the statements a little shorter. Remember that `x += delta_x` does exactly the same thing as `x = x + delta_x`.

Program 2.6: gutter.cpp (Version 2)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
        int i;
        double delta_x;
        double x;
        srand(time(NULL));
        x = 0.0;
        for ( i=0; i<10; i++ ) {
                delta_x = 100.0 * rand()/(1.0 + RAND_MAX);
                x += delta_x;
        }
        printf ( "%lf\n", x );
}
```

Exercise 12: Let's Race!

Modify your `gutter` program so that it looks like Program 2.6. Compile it, and then run it a few times. Each time you run it, you should see a single number, and you should get a different number each time (assuming you wait at least one second between tries, as before). Try racing your stone with your neighbors!

2.11. Tips for Using Loops

Almost all of the programs we write will use loops. Here are a few tips that will help keep you out of trouble when using them:

- **Count starting with zero, not one.** You could write a "for" loop like this to count from 1 to ten:

  ```
  for ( i=1; i<11; i++ )
  ```

  but you'll find later that it's more natural in C to number items starting with zero instead of one. So, in the programs we've been writing we loop ten times by writing a "for" statement like this, instead:

  ```
  for ( i=0; i<10; i++ )
  ```

  Doing it this way will make things much easier for you in the future.

- **Don't change the value of your counter variable inside the loop.** For example, what would this do?:

  ```
  #include <stdio.h>
  int main() {
          int i;
          for (i = 0 ; i < 10 ; i++) {
                  i = 100*i;
                  printf("loop number %d\n", i);
          }
  }
  ```

  (Note the line that reads `i = 100*i`.)

  If you tried it, you'd see that the program only prints out two numbers, instead of the ten numbers you might have expected. Why is this? It's because you've changed the value of `i` inside the loop.

  The first time around the loop, the program prints "0", and the second time around the loop it prints "100". So far, so good. But then the program stops.

  This happens because the value of `i` is now 100, so when we get back to the top of the loop, the "for" statement sees that "i<10" is no longer true, and the loop stops.[6]

- **Finally, don't assume that your counter variable has a useful value any place outside its loop.** After the loop is finished, does "i" contain the number of times around the loop, or something more or less? (Or even something completely different?) The answer can get complicated. It's better to assume that you can only trust the value of the counter variable when you're inside its loop.

[6] See the discussion about how "for" loops work in Chapter 1.

2.12. Nested Loops

Let's get back to our gutter program now. Imagine that we draw a starting line and arrange a bunch of our rocks behind it, ready to race each other down the gutter like racehorses in their starting gates. After many rainstorms, the rocks would all be at different locations somewhere lower down the gutter.

They'd be at different locations because each rock slides a different random amount during each rainstorm. A few rocks will get lucky and travel a long way. A few will travel unusually short distances. Most of the rocks will end up somewhere between these extremes, mounded up around some average distance.

Does the output of our program match this prediction? If we wanted something really boring to do, we could run Program 2.6 once for each rock, write down the results, and then graph them. Computers can save us that effort, though, and they're less likely to make the mistakes we might make while doing the work ourselves.

The inner loop of Program 2.7 is nested inside the outer loop, like these Russian Matryoshka dolls.

Source: *Wikimedia Commons*

We can modify our program so that it effectively runs the simulation many times. To do this, we'll need to add another loop. Take a look at Program 2.7.

Program 2.7: gutter.cpp (Version 3)

Changes from Program 2.6 are shown in bold.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
        int i;
        int j;
        double delta_x;
        double x;
        srand(time(NULL));
        for ( j=0; j<10000; j++ ) {
          x = 0.0;
           for ( i=0; i<10; i++ ) {
                delta_x = 100.0 * rand()/(1.0 + RAND_MAX);
                x += delta_x;
           }
          printf("%lf %d\n", x, j);
        }
}
```

Nested Loops

The new loop wraps around the loop that was already there. (We say that the old loop is "nested" inside the new loop.) Each time we go around the new loop we'll simulate another stone washing down the gutter for ten days. The variable `i` counts the number of rainstorms and `j` counts the number of stones. The program simulates 10,000 stones! That would be a lot of work by hand, but it's trivial for a modern computer.

The program prints out two numbers[7] for each stone: The total distance the stone travels, and the number of the stone's "starting gate". We number these gates from zero to 9,999, and use these numbers to keep track of which stone is which. We use the new variable `j` to represent the starting gate number, and this is the counter variable for the newly-introduced loop.

Each stone will start at the same place, so every time the program starts a new stone, it resets `x` (which represents the stone's position) to zero. When a stone has been through ten rainstorms, its final position and starting gate number are printed out, and then the program starts working on another stone.

Exercise 13: Scattering Stones

Modify your "gutter" program so that it looks like Program 2.7. Compile it, but don't run it like you've run the preceding programs. Instead, use the trick we saw in Chapter 1 that lets you send the program's output into a file, like this:

```
./gutter > gutter.dat
```

Now plot your results using *gnuplot*. Type *gnuplot*, then enter the following commands (can you guess what the *xrange* command does?):

```
set xrange [0:]
plot "gutter.dat"
```

You should see something like Figure 2.5. The horizontal axis shows how far each stone traveled. The vertical axis shows which gate the stone started from. As you can see, a "typical" stone travels about 500 cm, but some stones only make it to about 200 cm, and some go over 800 cm.

Does Figure 2.5 look the way we'd expect it to? Let's think about it. During each rainstorm, a stone travels a random distance between

[7] Notice that our `printf` statement here has two placeholders, "`%lf %d`", one for the stone's final position, which is a number containing decimal places, and one for the stone's starting gate, which is an integer.

zero and 100 centimeters. We'd expect the average distance to be 50 centimeters. So, after ten rainstorms, we'd expect a typical stone would travel $50 \times 10 = 500$ centimeters. This is the position of the densest part of Figure 2.5. A maximally sticky stone wouldn't move at all (travelling zero centimeters), and a maximally slippery stone would zip through a distance of $100 \times 10 = 1,000$ centimeters. We'd expect our graph to range from zero to 1,000 centimeters, with a peak at around 500 centimeters, and that's indeed what it shows.

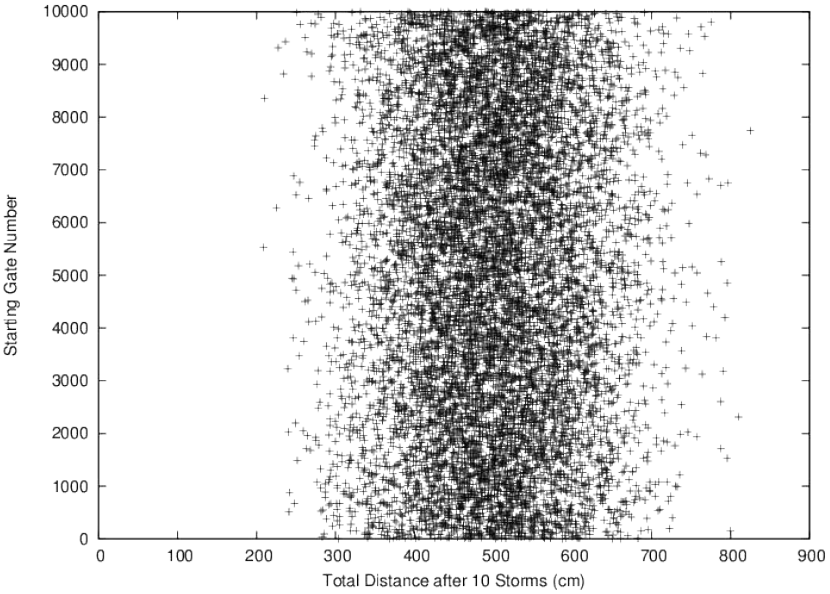


Figure 2.5: A plot of the results from our latest version of the "gutter" program.

But what about...?

Sometimes, you'll make a mistake that causes your program to keep looping forever. What can you do to stop this?

You can tell the program to stop running by pressing **Ctrl-C** (hold down the Ctrl key while pressing the "C" key).

Stopping a runaway program.

Source: Wikimedia Commons

2.13. Conclusion

Imagine that we continued to extend and improve our “gutter” program. We could add the effects of friction, rainstorms of random duration and strength, the slope of the gutter, and so forth. Eventually, we might have a program that could realistically simulate erosion, an avalanche or a mudslide.

Erosion near Bern, Switzerland

Source: Wikimedia Commons

For example, we could modify our program so that the range of random distances was determined by the duration of the rainstorm, instead of always being zero to 100 cm. Then we'd generate rainstorms of random durations and see what happens. By adding more and more refinements, we can make our simulation's results similar enough to reality to meet our needs.

Simulation programs like this allow us to handle large, complex problems by breaking them up into simple, understandable pieces. They represent an important computing technique that you can apply to many problems.

Practice Problems

1. As described in Section 2.6, write a program that prints out the value of `RAND_MAX`. Call your program `printrand.cpp`.

2. Write a program named `epoch.cpp` that prints the following:

   ```
   Seconds since 1970: ...
   Years since 1970: ...
   ```

   Where the `...` is replaced by the current number of seconds and years since 1970, based on the value returned by the `time` function, as described in Section 2.5. Check your program by running it several times to make sure that the number of seconds changes as time passes.
   Hint 1: The statement "`t = time(NULL);`" will store the number of seconds in the variable `t`.
   Hint 2: Assume that the number of seconds in a year is `60*60*24*365.25`.
   Hint 3: You'll probably want to use `%lf` as the placeholder when printing the year. Otherwise, *g++* might give you warnings or errors.

"Time keeps on slippin, slippin, slippin, into the future...." Steve Miller in 1977.
Source: Wikimedia Commons

3. Modify Program 2.4 so that it generates either a zero or a one. Then modify your new program so that it uses a loop to do this ten times. The resulting program does the equivalent of ten coin flips, with zero or one representing heads or tails. Call your new program `coinflip.cpp`.

4. Modify Program 2.4 so that it prints out two random digits between zero and nine. Make the program write the digits side-by-side, like `67` or `03`. Call your new program `percentile.cpp`. If you've ever played a roll-playing game like Dungeons and Dragons you've used ten- or twenty-sided dice to generate pairs of digits like this. In these games such a pair of dice are called *percentile dice*. The two digits they give you are interpreted as a percentage between 00% and 99%.

A 20-sided die shaped like an icosahedron. Two dice like this were originally used in Dungeons and Dragons for rolling percentiles. Later, they were replaced by two ten-sided dice. In this author's opinion, ten-sided dice are an abomination, since they aren't one of the five platonic solids!

5. Each line printed by Program 2.2 shows a single random integer. Using that program as an example, write a program that prints out *two* random integers on each line, separated by a space. Make the program print 10,000 pairs of integers. Let's call this program `tworand.cpp`. Use the trick you learned in Chapter 1 to send the program's output into a file named `tworand.dat`:

   ```
   ./tworand > tworand.dat
   ```

   Check the program's output by using *gnuplot* to plot the data in this file. Start *gnuplot* and give it the command:

   ```
   plot "tworand.dat"
   ```

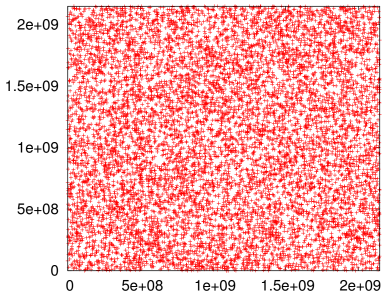


Figure 2.6: The output of the `tworand` program, plotted by *gnuplot*.

This causes *gnuplot* to use the two numbers on each line as the x and y coordinates of a point. You should see a graph that looks like Figure 2.6.

The `tworand` program generates a set of random points in the x, y plane. As we'll see later (in Chapter 10) this can be very useful.

6. Each line printed by Program 2.2 shows a single random integer. Using that program as an example, write a program that prints out *three* random integers on each line, separated by spaces. Make the program print 10,000 sets of three integers. Call this program `threerand.cpp`. Use the trick you learned in Chapter 1 to send the program's output into a file named `threerand.dat`:

```
./threerand > threerand.dat
```

Check the program's output by using *gnuplot* to plot the data in this file. Start *gnuplot* and give it the command:

```
splot "threerand.dat"
```

(**Note that the command is `splot`, not `plot`.** The `splot` command is used for plotting data in three dimensions.)

The command above causes *gnuplot* to use the three numbers on each line as the x, y and z coordinates of a point. You should see a graph that looks like Figure 2.7. As you can see, the numbers should evenly fill the box, since they come from a uniform distribution of random numbers (that is, any number is just as likely as any other number).

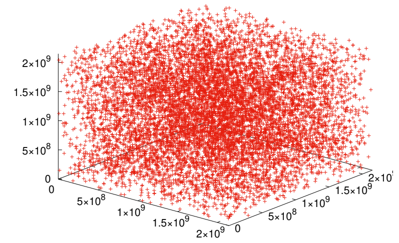


Figure 2.7: The output of the `threerand` program, plotted by *gnuplot*. Note that many versions of *gnuplot* will let you grab the image with your mouse and drag it around to see it from different angles. Try it!

7. Program 2.4 generates one random number between 1 and 6 every time you run it. Starting with this program, create a new program named `diceavg.cpp` that rolls a six-sided die 10,000 times and finds the average number that was rolled. Note that the average probably won't be an integer.

Hint 1: Add a couple of variables like this:

```
  int nrolls=10000;
  double sum = 0;
```

Hint 2: Add a loop that starts like this:

```
  for ( roll=0; roll<nrolls; roll++ ) {
```

Hint 3: Every time you go around the loop, do this:

```
     sum = sum + i;
```

8. Our gutter programs have a lot of numbers written into them: 10 days, 100 cm, 10,000 trials. If we want to change to, say, 1,000 trials,

we need to find all of the places in the program where we currently assume a value of 10,000, and change them.

It would be better if these numbers were more easily changed. Can you rewrite Program 2.7 so that the number of days, the maximum "slide" in cm, and the number of trials are given by variables defined near the top of the program?

For example:

```
int ndays = 10;
double maxslide = 100.0; // in cm.
int ntrials = 1000;
```

9. Using a nested pair of loops, as described in Section 2.12, write a program named grid.cpp that prints out the grid shown below:

```
[0,0] [0,1] [0,2] [0,3] [0,4]
[1,0] [1,1] [1,2] [1,3] [1,4]
[2,0] [2,1] [2,2] [2,3] [2,4]
[3,0] [3,1] [3,2] [3,3] [3,4]
[4,0] [4,1] [4,2] [4,3] [4,4]
```

Hint 1: Remember that you can leave off `\n` if you want `printf` to keep printing things on the same line.
Hint 2: It's perfectly OK to use `printf` to print nothing but a newline, like this: `printf ("\n");`

A grid like the one you produce in Problem 9 might be used to identify the squares on a Bingo card.
Source: publicdomainpictures.net

10. Make a new program named bingo.cpp that is a modified version of Program 2.4. The new program should be different from Program 2.4 in two ways: (1) The numbers it prints should be between 1 and 75, inclusive, and (2) instead of printing just one random number, it should use a pair of nested loops[8] to print a grid of random numbers, like a Bingo card. You could use this program to generate Bingo cards! See the two hints in Problem 9 for advice about how to print a nice-looking grid. Also, don't try to make a "Free Space" in the middle: Just put a number there, like all the other squares.

11. Imagine you have twelve 6-sided dice. Now roll all the dice at once and add up the numbers they show. This should give you a sum between 12 and 72. Write a program named 12dice.cpp that rolls twelve dice and prints their sum. Have the program repeat this 10,000 times. Run the program like this to send its output into a file named 12dice.dat:

```
./12dice > 12dice.dat
```

Now start *gnuplot* and give it the command `plot "12dice.dat"`. You should see a graph like Figure 2.8. Notice that the numbers

[8] See Program 2.7 for an example of nested loops.

tend to cluster around a value of 42. You might expect this, since the average value for rolling a single die is $(6+1)/2 = 3.5$ and $12 \times 3.5 = 42$.

Hint: Use a variable named `sum` to hold the sum of the 12 dice. Each time you start rolling the dice, remember to set `sum` to zero at the beginning. Then just add the value of each die to `sum` until you've added all twelve numbers. Print `sum`, then go on to the next roll.

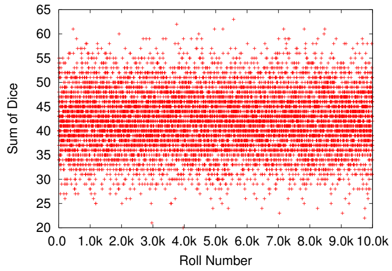


Figure 2.8: The sum of twelve dice, repeated 10,000 times. They cluster toward the center due to something mathematicians call the Central Limit Theorem. We'll talk more about this in Chapter 7.

12. Highly improbable things happen all the time. This might seem self-contradictory, but consider the following. Imagine that you're a golfer. You hit a golf ball and it lands on a blade of grass somewhere on the golf course (assume you didn't hit a sand trap, water hazard, or fellow golfer). That blade of grass is one among billions. The probability that you would hit that particular blade of grass is astronomically small. But it was certain that you would hit *some* blade of grass. This is sometimes called the "blade of grass paradox"[9].

Similarly, consider the case of a person who wins the lottery twice in one year. The odds against any particular person doing that are trillions to one. But, if you look at all of the lottery players in the U.S. over the course of a year, it's not too unlikely that *somebody* will win twice.

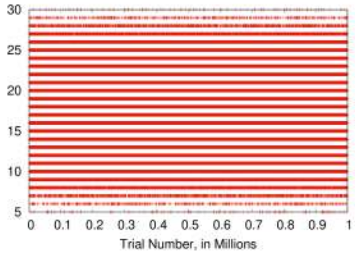


Figure 2.9: Output of the `improbable.cpp` program.

Let's try an experiment. Write a program named `improbable.cpp` that repeatedly rolls a set of 5 six-sided dice. Start out by copying Program 2.4, and then modify it so that, instead of rolling just one die, it rolls 5 dice, adds them up, and prints their sum. Then further modify the program so that it does this a million times. The result should be a program that prints a million numbers, with each number being the sum of 5 dice. (The program should end up having two nested "`for`" loops.) Then run the program like this:

```
./improbable > improbable.dat
```

and plot the data by giving *gnuplot* the following command:

```
plot "improbable.dat"
```

You should see a graph that looks like Figure 2.9. Notice the line of marks at the top, where the sum of the 5 dice is 30. These are the times when the program rolled five sixes. The odds against rolling five sixes are about 8,000 to one, but our program did it many times.

[9] See this article in the New York Times.

13. Imagine a person standing on a sidewalk. She's waiting for a friend who's late, and while she waits she paces back and forth. Sometimes she takes a step to the right, and sometimes she takes a step to the left. If we numbered the squares of the sidewalk, we could make a graph like Figure 2.10 that shows how her position changes over time.

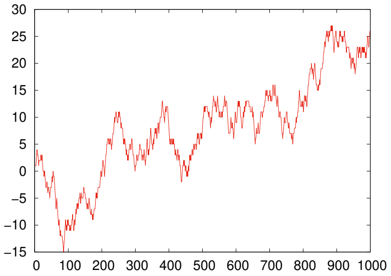
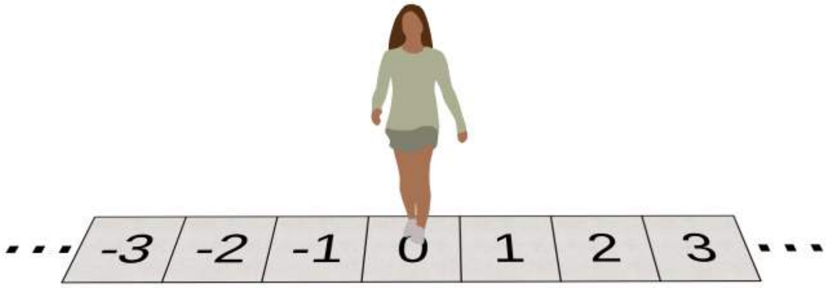


Figure 2.10: The person's position on the sidewalk is shown on the vertical axis. Time, in seconds, is on the horizontal axis. She starts at position zero and then wanders back and forth.



Let's say that during every second the person either (a) takes one step to the left, (b) takes on step to the right, or (c) stays where she is. Using Program 2.4 as a starting point, create a program named `wander.cpp` that simulates the person's motion over the course of 1,000 seconds.

Start by changing `min` and `max` to -1 and 1, respectively. That will cause the program to generate the numbers -1, 0, or 1, at random. We'll say that a step to the left is -1, a step to the right is 1, and staying in place is zero.

Then add two new variables:

```
int t;
int position=0;
```

`t` will keep track of the time, in seconds. `position` will keep track of which segment of the sidewalk the person is currently standing on. (We'll start her out at segment number zero.)

Finally, add a "`for`" loop to the program so that it generates 1,000 numbers instead of just one number. The loop should begin like this:

```
for ( t=0; t<1000; t++ ) {
```

Everytime around the loop, add a random -1, 1, or zero to the value of `position` and print the new value of `position`. When you run the program it should print one number on each line.

If you send the program's output into a file like this:

```
./wander > wander.dat
```

you can plot the output using the following command in *gnuplot*:

```
plot "wander.dat" with lines
```

The graph should look like Figure 2.10, although the pattern of ups and downs will be different every time you run the program.

This is an example of an important concept called a one-dimensional random walk. We often see data like this in the real world. Compare this graph to graphs of stock prices, for example. We'll be coming back to random walks again later.

3. *Writing Flexible Programs*

3.1. Introduction

The programs we've written so far have have all been designed to do one predetermined thing. If you wanted to change the behavior of one of these programs, you'd need to edit it and re-compile it. If you had to do this often, it would be rather inconvenient, and if you were a software vendor you almost certainly wouldn't ask your customers to edit and re-compile your program every time they needed to change a setting. (A vendor might not even want to *give* customers the source code for the program. Having the source code would allow the customers, or other vendors, to write their own programs, eliminating demand for your product!)

In this chapter, we'll see how you can write flexible programs that behave differently depending on input from the user.

In some situations, recompiling the program to change its settings isn't an option.

Source: Wikimedia Commons, Wikimedia Commons

3.2. Reading Input from the User

C provides a function called `scanf` that can read information typed by the person running your program. The `scanf` function causes your program to pause until the user has entered some information. After the information has been supplied, it's put into variables for later use, as illustrated in Figure 3.1.

Take a look at Program 3.1. This is a pretty useless program, but it illustrates how `scanf` works. When the program is run, it asks the user to enter a number[1], and then just tells the user what number was entered.

As you can see, the `scanf` function looks a lot like `printf`. The biggest difference is the ampersand ("`&`") in front of the variable `i`. For now,

[1] Remember that `printf` uses `%d` for `int` variables and `%lf` for `double` variables. You'll see that `scanf` does the same.

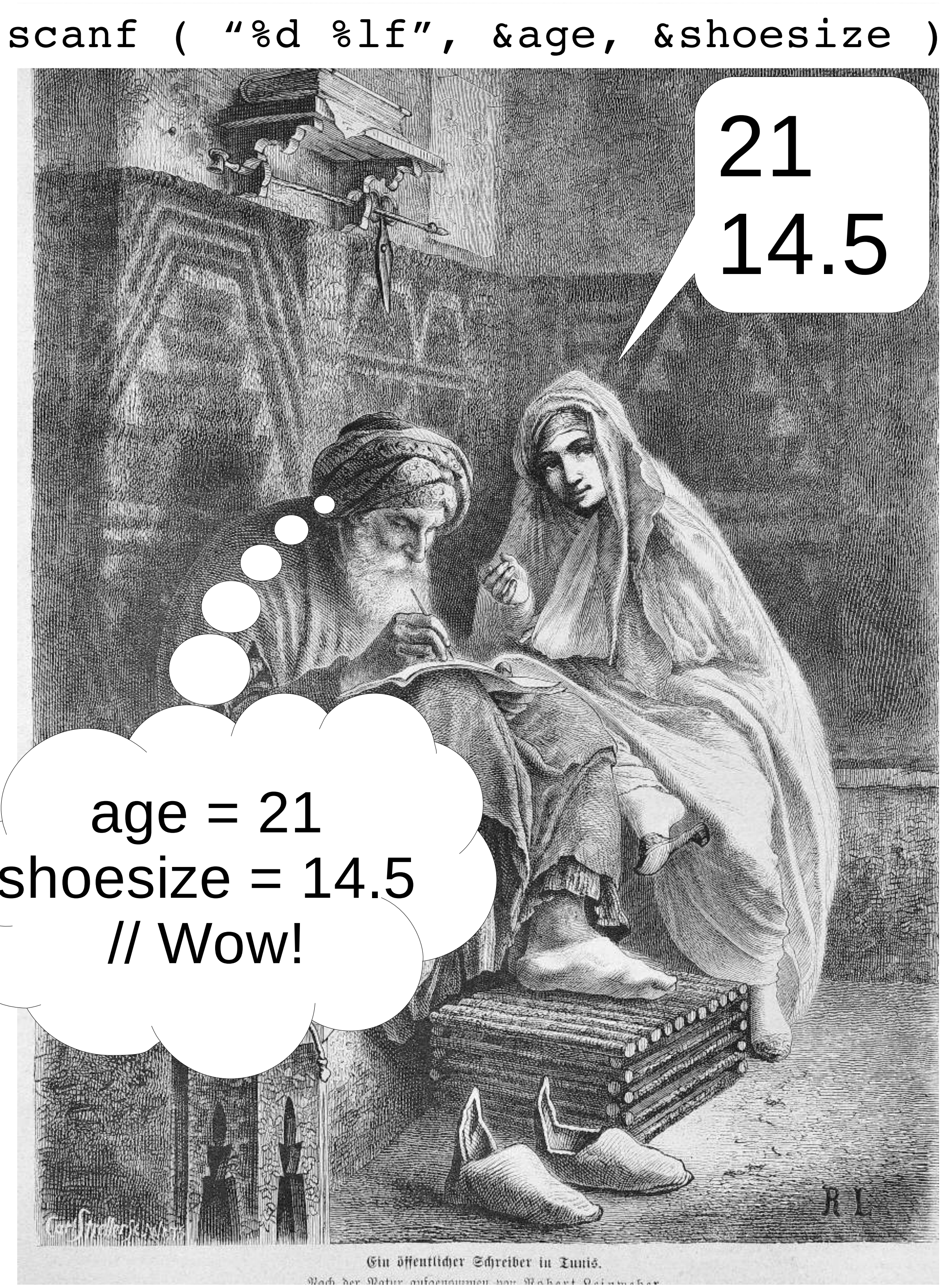

scanf (“%d %lf”, &age, &shoesize)
21
14.5
age = 21
shoesize = 14.5
// Wow!
R L
Ein öffentlicher Schreiber in Tunis.
Nach der Natur aufgenommen von Robert Leinweber

Figure 3.1: The `scanf` function acts like a scribe. It takes the information you give it and puts that information into variables in your program. We can only speculate about its internal commentary...

Source:Die Gartenlaube (1875), Wikimedia Commons

Program 3.1: reader.cpp

```
#include <stdio.h>
int main () {
  int i;

  printf("Enter an integer: ");

  scanf("%d",&i);

  printf("The number you entered was %d\n", i);
}
```

you don't need to understand why this ampersand is there, but you need to use it whenever `scanf` reads a number. We'll come back to it later and explain why.

Exercise 14: Using Scanf

Using *nano* and *g++*, create and compile Program 3.1. Be extra careful not to leave out the ampersand! Try running the program several times, giving it integers as input. Note that you'll need to press "Enter" after you've typed the number. Does the program work as expected?

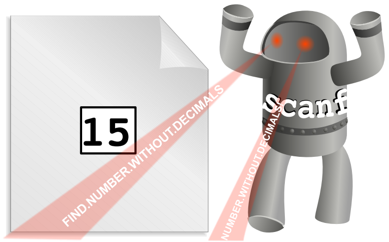
What happens if you enter spaces or tabs before or after the number? Does it make any difference?

Try giving the program a number with a decimal, like "1.5". What happens? What if you type extra text after the number, like "5 and other things"?

What happens if you type a letter as the first character?

Refer to Chapter 1 if you don't remember how to create and compile a program.

You can think of `scanf` as the opposite of `printf`. The `printf` function writes things, and the `scanf` function reads things. The "f" in both cases stands for "formatted", and both functions take a "format string" as their first argument. We've learned that the format string tells `printf` how to write its output. In the case of `scanf`, the format string tells the function what it should expect its input to look like.



`scanf` scans the text you type, looking for numbers (or other things) in a given format.

Source:Die Gartenlaube (1875), OpenClipart.org

`scanf` reads whatever the user types, then sorts it out and puts it into one or more variables. The format string we give scanf tells it what kind of input to expect, and how to sort it into the variables we specify.

In Program 3.1 we're only reading data into one variable. If we wanted to read the values of more variables, we could either add more `scanf` statements to the program, or we could use a format string like the one shown at the top of Figure 3.1, with more than one placeholder in it:

```
scanf ( "%d %lf", &age, &shoesize );
```

The number of placeholders in the format string must match the number of variables we give `scanf`.

When you give Program 3.1 a number like "1.5", you should see that it gets truncated to "1". This is because we told `scanf` to look for an integer by giving it the format string "`%d`". `scanf` stops looking as soon as it encounters something that doesn't look like part of an integer. If you enter "5 and other things", you'll see that the program thinks you typed "5".

3.3. `scanf` and Extra Spaces

As you saw in the exercise above, `scanf` ignores any leading or trailing spaces around placeholders. This is nice, because it makes your program forgive any extra spaces that the user might type.

For example, consider Program 3.2, which is just a modified version of Program 3.1. The new program asks the user to enter *two* integers. The format specification given to `scanf` is `"%d %d"`, meaning "look for one integer followed by some space and then another integer". (Remember that `%d` is a placeholder for an integer.) After the user enters the two numbers, they're put into the variables `i` and `j`. Finally, the program just prints the values stored in these variables to confirm that the program really got the numbers we tried to give it.

Program 3.2: reader.cpp, with 2 variables

```
#include <stdio.h>
int main () {
  int i;
  int j;

  printf("Enter two integers: ");

  scanf("%d %d",&i, &j);

  printf("The numbers you entered were %d and %d\n", i, j);
}
```

Exercise 15: Space Patrol

Create, compile and run Program 3.2, then try some experiments with it. The first time you run it, obediently give it two integers separated by a space. Then run it again, putting several spaces between the numbers. What happens if you press the "enter" or "return" key between the numbers instead of putting spaces? What about pressing "enter" or "return" multiple times?

You should find that the program behaves the same no matter which of these ways you choose to enter the numbers. As far as `scanf` is concerned, spaces, tabs, and returns are all the same thing, and it doesn't matter how many of them you enter. Programmers call these invisible characters "white space".

Roberta Leigh, producer of the 1960s British TV series *Space Patrol*. The show used puppets as its characters. The intrepid Captain Larry Dart sits to the left of Leigh.

3.4. Un-initialized Variables

When you enter a letter instead of a number, Program 3.1 behaves unexpectedly. Instead of a letter, the program might tell you that it saw some big number. It might even show you a different number if you do the same thing again. What's going on here? The problem is that `scanf` is looking for a number to put into the variable `i`, but it never sees one, so it doesn't change the value of `i`.

What value does `i` have if the program has never given it a value? Remember that each variable's value is stored in a chunk of the computer's memory[2]. When a program finishes, the computer can re-use that chunk of memory for another program. When a new program starts, the chunks of memory for all of its variables just contain whatever data was left over by the last program that used that space.

That's why Program 3.1 prints something unexpected if we enter a letter instead of a number. `scanf` never sets the value of `i` so the variable just has some leftover junk in it, which gets printed out by our `printf` statement. If we wanted to make things a little neater, we could change one line of the program so that it sets the value of `i` at the beginning of the program. Instead of

```
int i;
```

we could say

```
int i=0;
```

Then, if the user enters something that's not a number, the program

[2] See Section 1.12 in Chapter 1.

would always say that the number was zero. One lesson to learn from this is that you shouldn't assume that a variable has any value until you give it one. This will come up a lot later, so keep it in mind.

Later on (in Chapter 8) we'll talk about reading text. Until then, we'll only be using `scanf` to read numbers.

But what about. . . ?

What if we put text like "Hello World!" into the format string for `scanf`? Or what if we put a `\n` at the end of the format string?

First, if our program said `scanf("my age is %d",&i);` then we'd need to type something like "my age is 54", because the program would be looking for the text "my age is" followed by a number. Note that we wouldn't be allowed to have any extra spaces in front, either, since `scanf` only ignores extra spaces around placeholders like `%d`.

In the second case, `scanf` doesn't distinguish between space, tab, or newline characters. These are all "white space". When scanf sees white space in a format specifier, it waits for the user to type in any number of these characters, followed by at least one non-white-space character. If we said `scanf("%d\n",&i);` the program wouldn't continue until we'd entered a number, followed by one or more white space characters, followed by something that isn't a white space character.

3.5. Decisions, Decisions!

We've seen that computers are good at loops, but they're also good at making comparisons and decisions, and doing those things very rapidly.

Until now we've dealt with programs that follow a single predetermined path from start to finish. Now we'll look at ways to control the flow of our programs, making them do different things under different circumstances.

In C, you can use an "`if`" statement to make decisions. "`if`" statements check to see if some condition is true, then decide whether to take some action. Program 3.3 shows a straightforward example. The `printf` statement inside the curly brackets is only acted upon when the condition in the "`if`"'s parentheses is true. It's easy to read this as a sentence: "If i+j is greater than 10, print some stuff."

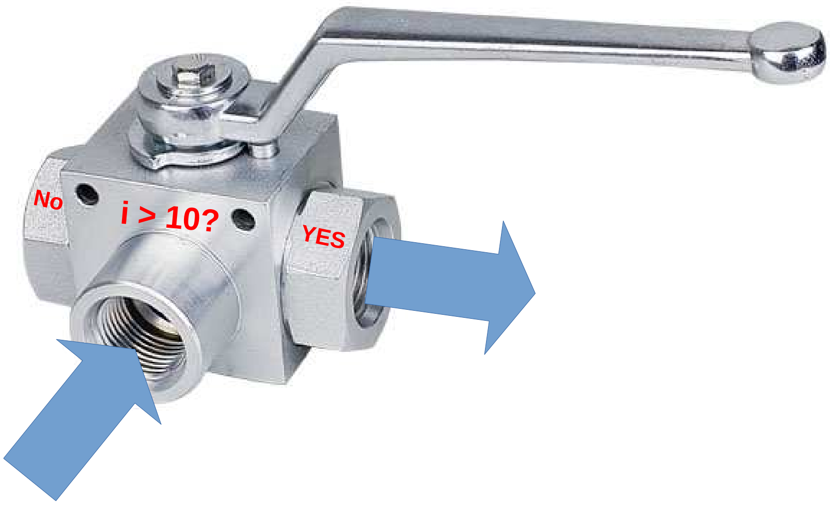


Figure 3.2: "`if`" statements are like valves that control the flow of your program.

Notice that Program 3.3 uses two `scanf` statements to read two numbers from the user.

Exercise 16: "`if`" Statements

Use *nano* to create Program 3.3, then compile it with *g++* and try running the program a few times. Does it behave as expected?

Program 3.3: checksum.cpp (Version 1)

```
#include <stdio.h>
int main () {
  int i;
  int j;

  printf("Enter an integer number: ");
  scanf("%d",&i);
  printf("Enter another  integer number: ");
  scanf("%d",&j);

  if ( i+j > 10 ) {
    printf("The sum is greater than 10\n");
  }

}
```

The most general form for an "`if`" statement looks like this:

```
if ( CONDITION ) {
  lIST OF THINGS TO DO
}
```

The "condition" is some test that will determine whether or not the following list of things should be done. We can check to see if two things are equal, or if one is greater than the other, or any of several other conditions. We can also combine several tests, and require (for example) that they all be true. Maybe we want to check to see if something *isn't* true. We can do that, too.

The "list of things to do" can include any C statements we want to use. This list is just a section of our program that will only be acted upon when "`if`" statement's condition is met.

Here’s another example of an “`if`” statement:

```
if ( i > 10 ) {
        printf ("i is greater than 10.\n");
        printf ("The value of i is %d\n", i);
}
```

You can also nest “`if`” statements, as in this example:

```
if ( a < 5 ) {
  printf ("a is less than 5.\n");
  if ( b > 100 ) {
        printf ("and b is greater than 100.\n");
  }
}
```

In the nested example, the `printf` statement inside the second “`if`” would only be acted upon if both `b > 100` and `a < 5` are true statements.

3.6. True or False?

The computer looks to see whether the statement in parentheses after “`if`” is true. Is `a` really less than five? Is `b` really greater than 100?

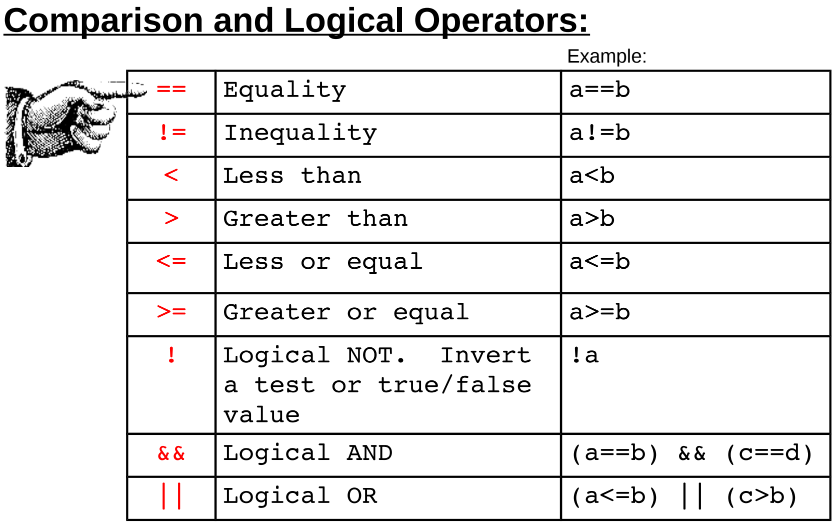
The C language provides several comparison operators that can be used in “`if`” statements. We’ve already seen the “<” operator in the loops we’ve written in earlier chapters, where it appears in expressions like `for (i=0;i<10;i++)`. In Program 3.3 above, we see the “>” operator.

Sometimes we want to combine multiple comparisons, like “this and that” or “this or that”. Maybe we even want to require “this but not that”. For these purposes, C provides a set of logical operators. The “and” operator (“`&&`”) can be used to say things like

```
if ( (a<6) && (b>3) ) {
   printf ( "Do stuff.\n");
}
```

meaning “If `a` is less than 6 *and* `b` is greater than 3, do stuff”.[3] The “or” operator (“`||`”) can be used in expressions like “`(c<5)||(d<5)`”, meaning “either `c` is less than 5 *or* `d` is less than 5”. An exclamation point in front of an expression means “not”. For example, “`!(a>10)`” means “`a` is *not* greater than 10”. Figure 3.3 shows C’s comparison and logical operators.

[3] Note how we use parentheses here to enclose each simple expression, and then put the whole expression inside the “`if`” statement’s “(CONDITION)” parentheses.



Comparison and Logical Operators:

		Example:
==	Equality	a==b
!=	Inequality	a!=b
<	Less than	a<b
>	Greater than	a>b
<=	Less or equal	a<=b
>=	Greater or equal	a>=b
!	Logical NOT. Invert a test or true/false value	!a
&&	Logical AND	(a==b) && (c==d)
||	Logical OR	(a<=b) || (c>b)

Figure 3.3: These operators are particularly useful in "if" statements. They compare values, or do logical operations like "and" or "or". Pay particular to ==, as described in the next section.

3.7. Testing Equality

Note in particular the "==" operator in Figure 3.3. This is the source of a lot of confusion. This operator *compares* two values to see if they're equal. This is often confused with "=", which *assigns* a value to a variable.

You can use the == operator in an "if" statement to compare two values. For example:

```
if ( i == 5 ) {
  printf ("Do stuff.\n");
}
```

would mean "If i is equal to 5, do stuff".

In C, if I say "a==2" I'm saying "compare the value in 'a' with the value '2' and tell me if they're the same." On the other hand, if I say "a=2" I'm telling the program to stick the value "2" into the variable "a". The most important thing to remember is that the "==" operator doesn't change the values of the variables, but the "=" operator does. This confusion results in many bugs.

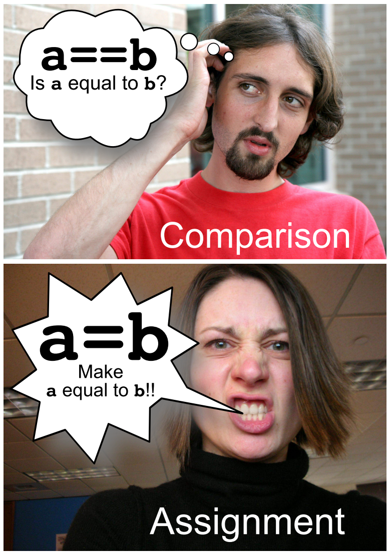


Figure 3.4: Use == to test equality, and = to force equality.

Source: Wikimedia Commons 1, 2

But what about. . . ?

What would happen if you mistakenly used "`i = 5`" instead of "`i == 5`" in an "`if`" statement?

To answer that, we first need to think about how the computer interprets these conditions. As it turns out, the the computer actually converts everything inside an "`if`" statement's "(CONDITION)" to a number. If the number is zero, the condition is false. If it's not zero, it's true. This means that an expression like

```
if ( 1 ) {
  printf ("Do Stuff.\n");
}
```

would cause "Do Stuff" to always be printed, since the number 1 is (and always will be) different from zero.

Sometimes programmers take advantage of this. We can have an "`if`" statement look at the value of a variable, and only act if the variable has a non-zero value. The expression `if ( width )` would only be acted upon if the variable "`width`" had a non-zero value, and `if ( !width )` would only be acted upon if "`width`" was equal to zero.

Now back to the question at hand: What if we accidentally wrote `if ( i=5 )` instead of `if ( i==5 )`? Remember that "`i=5`" means "assign the value 5 to the variable `i`". Would doing this inside the "(CONDITION)" of an "`if`" statement give a true or a false result? Perhaps surprisingly, it depends on what value we assign to `i`. If we say `if ( i=0 )` the result will always be false. If we use any other value (non-zero), the result will always be true.

That's because, in C, the numerical "value" of "`i = 5`" is just the value of `i`. So, the expression `i = 0` will always be false, but `i =` (anything else) will be true.

If you find that your program is acting as though an "`if`" condition is always true or always false, even though you think it shouldn't be, check to make sure you haven't used = where you should have used ==. Even though *g*++ will allow you to use = in an "`if`" condition, you should never use it there.

Program 3.4: checksum.cpp (Version 2)

```
#include <stdio.h>
int main () {
  int i;
  int j;

  printf("Enter an integer number: ");
  scanf("%d",&i);
  printf("Enter another  integer number: ");
  scanf("%d",&j);

  if ( i+j > 10 ) {
    printf("The sum is greater than 10\n");
  } else {
    printf("The sum is NOT greater than 10\n");
  }

}
```

3.8. Choosing Between Several Alternatives

"Good banana, bad banana..." (Women sorting bananas in Belize)

Source: Wikimedia Commons

Take a look at Program 3.4, which is just a slightly modified version of Program 3.3. As you can see, you can optionally add an "else" clause to an "if" statement. If the condition in parentheses is false, the actions in the "else" clause will be done.

Exercise 17: ...Or Else!

Modify your checksum.cpp program so that it looks like Program 3.4. Compile it, then run it several times. Make sure you give it some pairs of numbers that add up to more than ten, and some that have a sum smaller than ten. Does your program behave as expected?

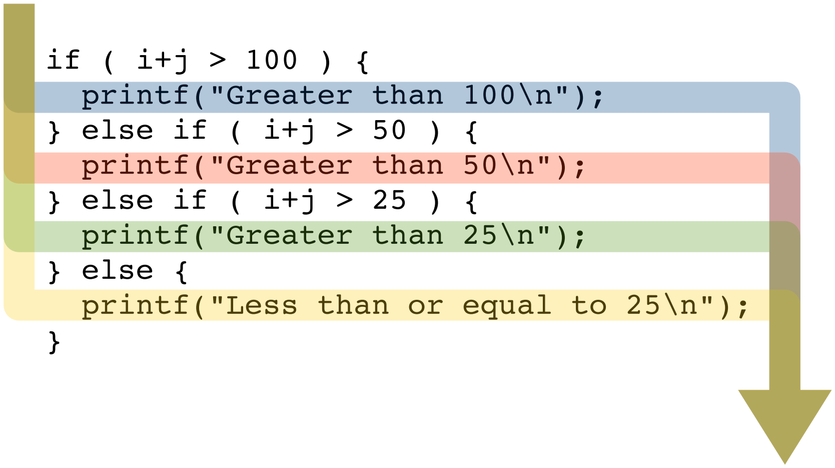
You can add as many other options as you want, using "`else if`" clauses:

```
if ( i+j > 100 ) {
  printf("The sum of these numbers (%d) is greater than 100\n", i+j);
} else if ( i+j > 50 ) {
  printf("The sum of these numbers (%d) is greater than 50\n", i+j);
} else if ( i+j > 25 ) {
  printf("The sum of these numbers (%d) is greater than 25\n", i+j);
} else {
  printf("The sum of these numbers (%d) is less than 25\n", i+j);
}
```

Each "`else if`" has some alternative condition that may be satisfied.

If nothing else is true, the statements in the final, "else", clause are acted upon. [4]

Only the *first* true condition will be acted upon. Even if other later conditions are true too, they'll be ignored. If you have a final "else" statement in the list, that will only be acted upon if *none* of the "if" or "else if" conditions are met. You don't need to have an "else" section. Without it, the "if" statement will just do nothing when none of the conditions are true.



```
if ( i+j > 100 ) {
  printf("Greater than 100\n");
} else if ( i+j > 50 ) {
  printf("Greater than 50\n");
} else if ( i+j > 25 ) {
  printf("Greater than 25\n");
} else {
  printf("Less than or equal to 25\n");
}
```

Figure 3.5: An "if" statement creates a set of alternative paths that the computer can follow when walking through your program.

When the computer runs one of your programs, you might imagine the computer starting at the top of the program and walking through it, line by line, until it gets to the bottom. Up until now, the programs we've written have only had one possible path. The "if" statement gives the computer multiple alternative paths it can follow.

Exercise 18: More Choices

Once again modify your checksum.cpp program. This time, add "else if" sections to your "if" statement so that the program tells you whether the sum is greater than 100, greater than 50, greater than 25, or less than 25, as shown in the examples above. Run the program several times, giving it different pairs of numbers so that you test each possible path through the "if" statement.

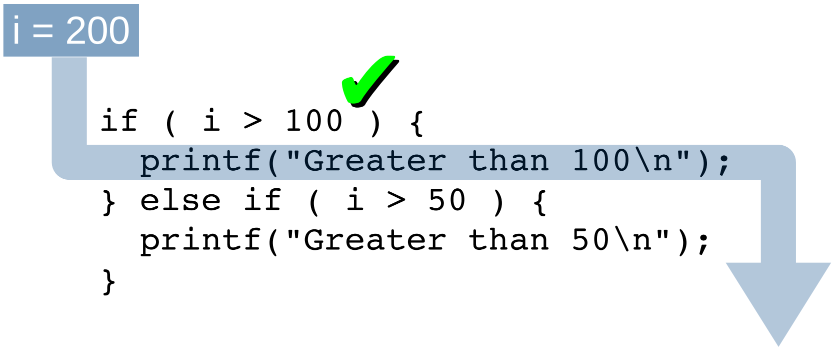
[4] Notice that even these complicated "if" statements can still be read as sentences: "If this is true, do something. Otherwise, if that is true do a different thing, ...".

3.9. "if/else if" versus multiple "if" statements

It's important to realize that an "if" statement always says "Here are some options. Do the first one that matches." The "if", "else if", and "else" lines in Figure 3.5 are all part of one unified statement that defines the options and tells the computer how to choose between them.

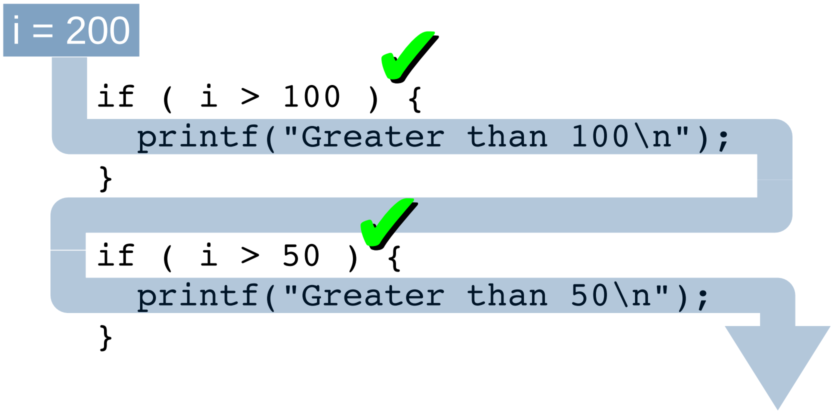
You might be tempted to use several independent "if" statements instead of one big "if/else if" statement, but you should remember that these are different.

You can see this difference in the examples shown in Figures 3.6 and 3.7. The first example shows a single "if/else if" statement that chooses between two options. The second example shows two independent "if" statements.



```
if ( i > 100 ) {
  printf("Greater than 100\n");
} else if ( i > 50 ) {
  printf("Greater than 50\n");
}
```

Figure 3.6: If i=200, this statement will print "Greater than 100" and nothing else. Only the first matching option is acted upon in an "if/else if" statement.



```
if ( i > 100 ) {
  printf("Greater than 100\n");
}

if ( i > 50 ) {
  printf("Greater than 50\n");
}
```

Figure 3.7: Alternatively, this *pair* of "if" statements will print "Greater than 100" followed by "Greater than 50", since both are true and the two "if" statements are independent.

Keep this in mind when you're writing programs that need to choose one option out of several possibilities.

But what about. . . ?

If you look at other people's C programs you might see "`if`" statements like this:

```
if ( i == 5 )
        printf ( "i is equal to 5\n" );
```

Notice that there are no curly brackets here. This is different from the "`if`" statements we looked at above.

The C language allows you to omit the curly brackets if there's only one line in the list of statements controlled by an "`if`" statement. This can make your program shorter, but I don't recommend that you do this, because it can lead to confusion later.

Consider what would happen if you used a line like the one above, and later modified the program by adding another line, like this:

```
if ( i == 5 )
        printf ( "i is equal to 5\n" );
        printf ( "Do some other stuff\n");
```

You might mistakenly think that the new line is also part of the "`if`" statement, but it's not. The new `printf` statement will always be executed, no matter what the value of `i` is.

This is exactly what led to a scary security bug (called the "Goto Fail" bug) on Apple computers in 2014.

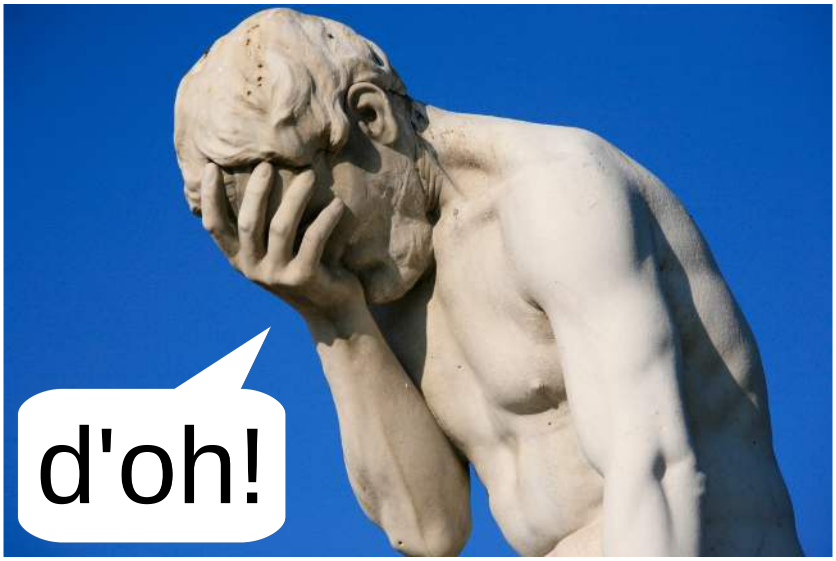


Figure 3.8: Sticking to a well-chosen programming style can help prevent errors in your programs.

Source: Wikimedia Commons

3.10. Using "and" and "or"

Sometimes we want to check more than one thing in an "`if`" statement. For example, imagine that you are enrolled in a class that has both a written and an oral exam. To pass the course, you need to get a passing grade on *both* exams. If the teacher wrote a program to tell her which students passed, it might include an "`if`" statement like this:

```
if ( written >= 70 && oral >= 55 ) {
    printf ("Student passed! :-)\n");
} else {
    printf ("Student failed. :-(\n");
}
```

The `&&` in the "`if`" statement means "**and**". This statement says that the student passes the class if they get a score greater than or equal to 70 on their written exam **and** they get a score greater than or equal to 55 on the oral exam.

Alternatively, we could re-write the statement like this:

```
if ( written < 70 || oral < 55 ) {
    printf ("Student failed. :-(\n");
} else {
    printf ("Student passed! :-)\n");
}
```

Here we're using `||`, which means "**or**". The statement now says that if the student got a written score less than 70 **or** an oral score less than 55, they **failed**.

Augustus de Morgan, one of the founders of modern mathematical logic.

Source: Wikimedia Commons

There's an important principle in the mathematics of logic that's called de Morgan's theorem. It says you can always rewrite a logical condition by replacing "and" with "or" and flipping everything to its opposite. That's what we've done in going from the first example above to the second. If you go on in programming, or into a field like digital circuit design, you'll find de Morgan's theorem very useful. Sometimes it can make tangled logical expressions a lot simpler.

You might be wondering about the order of operations in these "`if`" statements. There are a lot of things going in in an expression like "`written >= 70 && oral >= 55`". In what order does the program do these things? Do we need to add parentheses?

Expressions like this are evaluated in a well-defined order that's an extension of the "PEMDAS" rule you probably learned in school[5]. Consider this expression:

```
if ( 2*x+5 < 10 && y*6-3 > 4 ) {
```

The PEMDAS rules would tell us to multiply `2*x` first and then add `5`. Similarly, we'd multiply `y*6` and then subtract `3`. In C, comparison operators like < and > come after PEMDAS, so the next thing we'd do is check to see if `2*x+5` is less than `5`, and then check to see if `y*6-3` is greater than `4`. Finally, we'd deal with the logical operators like `&&` and `||`, so we'd check to see if `2*x+5 < 10` **and** `y*6-3 > 4`.

To help you remember this, you might just tack a "CL" on the end of PEMDAS, for "Comparison" and "Logic", to make PEMDASCL (rhymes with "rascal"!)[6].

3.11. Catching Mistakes

There's a way to know that the user has mistakenly entered something that's not what `scanf` expects. The `scanf` function returns a value, just like the functions we've seen in math class. The value returned by `scanf` is the number of things that it successfully read. We can use this value in an "`if`" statement to check whether `scanf` was able to understand what the user typed.

For example, in Program 3.3 (`checksum.cpp`) we could add another "`if`" statement to check for mistakes:

```
printf("Enter two integer numbers: ");
if ( scanf("%d %d",&i, &j) == 2 ) {

  if ( i+j > 10 ) {
    printf("The sum is greater than 10\n");
  }

} else {
  printf("Please try again.\n");
}
```

The new "`if`" statement checks to see if the value returned by the `scanf` function was 2, meaning that it had successfully read two numbers. If this weren't the case, then the "`else`" would cause the program to print an error message.

For simplicity, we'll leave out this check in the following examples in this chapter, but we'll use it later.

[5] PEMDAS says to do things in this order: Parentheses, Exponentiation, Multiplication, Division, Addition, Subtraction.

[6] You can find the full order of operations (called "operator precedence") for C here:
https://en.cppreference.com/w/c/language/operator_precedence

3.12. Writing a Math Quiz Program

Now let's do something more practical. Take a look at Program 3.5.

Albert Anker, *Mädchen mit Schiefertafel*

Source: Wikimedia Commons

Program 3.5: mathquiz.cpp (Version 1)

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
int main () {
  int i;
  int j;
  int sum;

  srand(time(NULL));

  i = (int)( 100.0 * rand()/(1.0 + RAND_MAX) );
  j = (int)( 100.0 * rand()/(1.0 + RAND_MAX) );

  printf("What is %d + %d ?: ", i, j);
  scanf("%d",&sum);

  if ( i+j == sum ) {
    printf("Right!\n");
  } else {
    printf("Nope. The sum of these numbers is %d. Go back to school.\n",
           i+j);
  }
}
```

Program 3.5 is a simple math quiz program. It generates two random integers between zero and 100, and asks the user to add them and enter the sum. The program then checks to see if the user got it right.

Exercise 19: Making a Math Quiz

Create Program 3.5. Be careful of all the parentheses, and make sure you have all of the necessary semicolons. Run the program several times. Are you a math wizard?

Notice how we've written the statements with `rand` in them. We want our random numbers to be an integers[7], so this is a little different from what we did in Chapter 2, where we wanted to generate random distances that could contain decimals. In Program 3.5 we convert our random numbers into integers by enclosing them in `(int)(...)`[8]

[7] You'll see why later in this chapter, when we talk about comparing numbers with decimal places.

[8] Programmers call this kind of thing "casting". In this case, we're "casting our number as an `int`". Think of it as casting an actor in a different role. Here, we're taking a number that would otherwise be a `double` and casting it as an `int`.

The program uses the `scanf` function to read input from the user. Then, in the program's "`if`" statement we use the == operator to see if the number entered by the user (`sum`) equals the actual sum of the two random integers (`i+j`). If the user gets it wrong, the program prints out the real sum.

3.13. A Longer Math Practice Program

What if we wanted our program to keep asking us questions? We could just add a loop to it.

In Program 3.6 we take the integer addition program we made before, and wrap it with a loop. The loop keeps the program asking questions until we've answered ten of them.[9]

The only differences between Programs 3.5 and 3.6 are the new variable `nproblems`, to count the number of questions asked, and the "`for`" loop.

Hong Kong children demonstrating their math skills.

Source: Wikimedia Commons

Program 3.6: loopquiz.cpp

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
int main () {
  int i;
  int j;
  int sum;
  int nproblems;
  srand(time(NULL));
  for ( nproblems = 0; nproblems < 10; nproblems++ ) {
          i = (int)( 100.0 * rand()/(1.0 + RAND_MAX) );
          j = (int)( 100.0 * rand()/(1.0 + RAND_MAX) );
          printf("What is %d + %d ?: ",i,j);
          scanf("%d",&sum);
          if ( i+j == sum ) {
            printf("Right!\n");
          } else {
            printf("Nope. The sum is %d. Go back to school.\n", i+j);
          }
  }
}
```

Think about how you might modify Program 3.6 to make it even better. Could you make the program keep score, and print out the score at the end? Could you use an "`if`" statement and random numbers to make the program choose addition or subtraction at random?

[9] If the user gets tired before answering all of the questions, Ctrl-C can be used to stop the program.

Exercise 20: A Better Quiz

So far, we've used the commands *nano*, *gnuplot*, *g++*, and *ls* (for showing a list of files). Let's use another command now. The *cp* command makes a copy of a file. Use it to make a copy of your `mathquiz.cpp` file by typing the following:

```
cp mathquiz.cpp loopquiz.cpp
```

The command above will make a new file called `loopquiz.cpp` that's a copy of your `mathquiz.cpp` file.

Now use *nano* to modify `loopquiz.cpp` so that it contains the changes shown in Program 3.6. Compile the program with *g++* and run it. Does it behave as it should?

Figure 3.9: Albert Anker, *Die Dorfschule von 1848*

Source: Wikimedia Commons

3.14. Comparing Numbers with Decimal Places

In our math quiz programs we've used integer numbers. What if we had used numbers with decimal places instead? Consider Program 3.7, which is just like Program 3.5, except that we've changed all of the integers into numbers with decimal places[10].

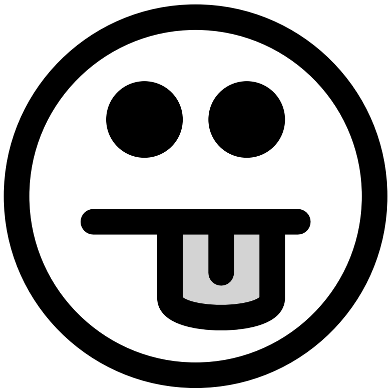
If you tried using this program, you might be surprised by what it does. Here's what it might look like:

```
What is 30.345296 + 60.080443 ?: 90.425739
Nope. The sum of these numbers is 90.425739. Go back to school.
```

But we got the sum right, didn't we? The program even tells us so! Why doesn't it work as expected?

Program 3.7: Why doesn't this work?

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
int main () {
  double i;
  double j;
  double sum;

  srand(time(NULL));

  i = 100.0 * rand()/(1.0 + RAND_MAX);
  j = 100.0 * rand()/(1.0 + RAND_MAX);

  printf("What is %lf + %lf ?: ", i, j);
  scanf("%lf",&sum);

  if ( i+j == sum ) {
    printf("Right!\n");
  } else {
    printf("Nope. The sum of these numbers is %lf. Go back to school.\n",
           i+j);
  }
}
```



The reason has to do with the difference between numbers with decimal places (which can have digits that go on forever – think of π, for example) and integers, which always have a finite number of digits.

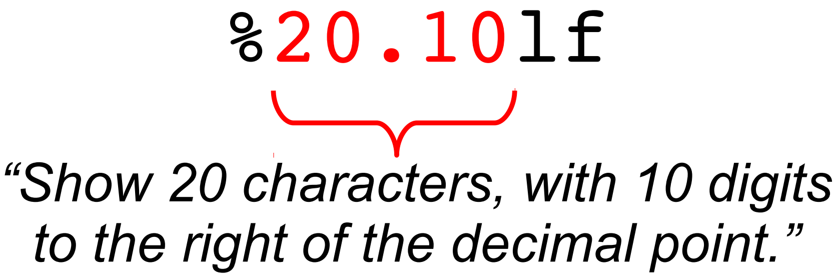
[10] We changed `int` to `double` and `%d` to `%lf`, and we omitted the `(int)(...)` when generating our random numbers.

When `printf` prints a `double`, it rounds the number off after a few decimal places. When you use the `%lf` format to print out a number, your program shows the first six decimal places, but inside the program the number is actually much more precise.

If we tell printf to show us more decimal places, we'll see what went wrong above. We can do so by modifying the `%lf` placeholder in our printf statement.

Instead of `%lf`, we can write an expression like `%`**`x.y`**`lf`, where `x` is a number that tells the program how much space to reserve for printing out the number, and `y` is a number that says how many digits to the right of the decimal point should be printed. We can leave off either `x` or `y` and `printf` will try to figure out what's the best thing to do on its own.

For example:


%20.10lf
"Show 20 characters, with 10 digits to the right of the decimal point."

If we had replaced `%lf` with `%.10lf` in the last printf statement of Program 3.7 (to print ten decimal places instead of the normal six) we would have seen:

```
What is 30.345296 + 60.080443 ?: 90.425739
Nope. The sum of these numbers is 90.4257384084. Go back to school.
```

As you can see, the number the computer was thinking of really didn't match the number we typed.

3.15. The Right Way to Do It

The right way to compare numbers with decimal places is to ask whether they differ by more than some small amount, which we'll call "`epsilon`".

In Program 3.8, we define epsilon to be something acceptably small for our purposes, and then we use the "`fabs`" function[11] to get the absolute value of the difference between the actual sum and our guess. If this difference is less than `epsilon`, we say we're close enough.

To use the `fabs` function, you'll need to add `math.h` at the top of your program.[12]

Program 3.8: The Right Way

```
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
int main () {
  double i;
  double j;
  double sum;
  double epsilon = .000001;

  srand(time(NULL));

  i = 100.0 * rand()/(1.0 + RAND_MAX);
  j = 100.0 * rand()/(1.0 + RAND_MAX);

  printf("What is %lf + %lf ?: ", i, j);
  scanf("%lf",&sum);

  if ( fabs(i+j - sum) < epsilon ) {
    printf("Right!\n");
  } else {
    printf("Nope. The sum of these numbers is %lf. Go back to school.\n",
           i+j);
  }
}
```

This is the *right* way to compare numbers with decimal places.

3.16. Conclusion

This chapter has covered a couple of tools you can use to allow users to control your program. The `scanf` function lets your program get input from the user, and "`if`" statements let you program make decisions. Combine these new tools with the elements of C you've learned in earlier chapters (loops, random numbers, *et cetera*, and you can already create some pretty sophisticated programs.

[11] We'll learn more about C's math functions in Chapter 4.

[12] Note that we could have done the same thing without `fabs` by checking to see if the difference was somewhere between `-epsilon` and `epsilon`.

Practice Problems

Source: Wikimedia Commons

1. Using Program 3.1 as an example, write a program that asks you for a circle's radius and then tells you the area of the circle. Use 3.14 as the value of π, and remember that the area of a circle is πr^2. Make sure the program tells the user that this is the area (don't just print a number without anything else). Call your program `circlearea.cpp`. **Hint:** You'll want to be able to enter numbers like "1.5" as the radius, so you'll need to use a `double` variable, not an `int`.

2. If you throw a ball straight up into the air with an initial velocity v it will reach a height of
$$\frac{v^2}{2g}$$
where $g = 9.8\ m/s^2$, the acceleration of gravity near the earth's surface. Write a program named `playball.cpp` that asks the user to enter the ball's initial velocity (in meters per second), and tells you how high the ball would go (in meters). Make sure your program tells the user what units to use when entering the velocity, and what units are used when reporting the height. (**Hint:** A ball thrown with a velocity of 10.5 m/s should reach a height of about 5.6 meters. Use this to check your program.)

3. Modify the looping version of the math quiz program (Program 3.6) so that it asks the user how many math problems he/she wants to answer. Use `scanf` to put this number into an integer variable, and use that variable in the program's "`for`" statement to control how many times the program loops. Call the new program `nloop.cpp`.

Source: Wikimedia Commons

4. Write a program named `airflow.cpp` that asks you for the length, width, and height of a rectangular room, in feet. Inside the program, calculate the volume of air in the room. Assume we'd like to replace all of the air in the room ten times per hour. That would mean we need to remove 1/6 of the room's air every minute. Fans are typically rated in terms of the number of cubic feet per minute that they can move. Have your program tell us how many cubic feet per minute we need to move in order to replace the room's air ten times per hour.

5. Write a program named `checkage.cpp` that asks the user for his/her birth year (like "1998") and the current year (like "2017"). Use an "`if`" statement to tell the user if he/she is under 21 years old, or not. (Ignore the birth month, and assume that everyone was born on January 1. Include people who are exactly 21 in the "not under 21" group.)

6. Write a phone menu program. Start by printing the following menu and asking the user to enter one of the numbers:

1	For sales
2	For billing
3	For support
4	For a live human being

   Use `scanf` to read the number entered by the user. Make an "`if`" statement like the one on Page 90, using "`else if`", and have it print out an informative message for each of the possible choices. (For example, "You have reached the sales department.") Use an `else` statement to give the user an informative message if she/he enters a number that's not on the menu. Call your program `phonemenu.cpp`.

William Howard Taft, 27th President of the United States.

Source: Wikimedia Commons

7. Modify the looping version of the math quiz program (Program 3.6) so that it keeps score, and tells the user how well he/she did at the end. (That is, print out a message like "You got 8 out of 10 answers right!") Call your new program `mathscore.cpp`.

8. Modify Program 3.6 so that it randomly picks addition or subtraction for each problem.

   Hints:

   - Look back at Program 2.3 in Chapter 2 to see how to generate a random number between zero and one.
   - Check to see whether this random number is greater than 0.5. If it is, choose addition. If it's not, choose subtraction.

A 2x4 driven through a palm tree in Puerto Rico by a 1928 hurricane.

Source: Wikimedia Commons

9. Hurricanes can hurl objects with tremendous force. Homeowners sometimes nail sheets of plywood over windows in preparation for a major storm. Studies done at Clemson University[13] have looked at the effect of 2x4 pieces of lumber fired with various velocities at plywood sheets. They found that the thickness of plywood required to stop such a projectile was proportional to the projectile's momentum. In mathematical terms, they found that the thickness required to stop the projectile was $t = 0.00032 \times m \times v$, where t is measured in meters, m is the mass of the projectile in kg, and v is the projectile's velocity in m/s.

   Write a program named `2x4.cpp` that asks the user to enter a velocity in meters per second. Have the program calculate t from the equation above, using 9.45 kg for the mass of the projectile (that's approximately the mass of a ten-foot pressure-treated pine 2x4). Have the program tell the user what thickness, in meters, of plywood they'll need to protect their home from such a projectile

[13] See https://www.fema.gov/previous-missile-impact-tests-wood-sheathing

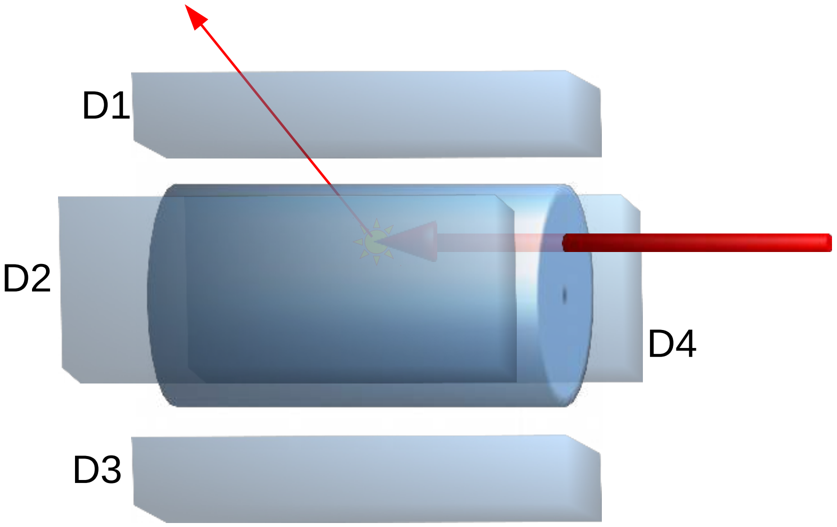
flying at this velocity. Also tell them what this thickness is in inches, by mutiplying the thickness in meters times 39.37. Test your program by telling it the velocity is 28 m/s (which is about 100 kilometers per hour). It should tell you that the required thickness of plywood is about 3 inches.

10. Write a program named `ridecheck.cpp` that checks to see if the user is eligible to ride a roller coaster. The program should ask the user for her height, in feet, and age, in years. Assume that the height might have a decimal place (like 4.9) but assume that the age will be an integer (like 21). If the user's age is greater than 11 **and** her height is greater than 4.5 feet, the program should say that she's allowed to ride. Otherwise, the program should say "Sorry, you're not allowed to ride.". Don't use more than one "`if`" statement in your program.

Are you tall enough? (Illustration by John Tenniel for Lewis Carroll's *Alice in Wonderland*.

Source: Wikimedia Commons

11. You're a physicist working at CERN, and your experiment uses the apparatus shown below. In the middle there's a cylindrical target, at which you'll be shooting a beam of particles. Some of the particles entering the target will decay while inside, and emit other particles. Each emitted particle will shoot out of the cylinder and go through one of four rectangular detectors arranged around the target. The detectors are named D1, D2, D3, and D4, and each one measures the energy of particles passing through it. You want to check periodically to see whether any of the four detectors saw a particle.



Write a program called `4signal.cpp` that asks the user to enter four energy values (numbers that might contain decimal points), one for each of the four detectors. Use a single "`if`" statement to see if any of the values was greater than 100. If so, the program should print "Saw a particle." Otherwise it should print "No particles this time."

12. Write a program named `spherestats.cpp` that lets you enter a sphere's radius, then prints a menu of options:

```
    What do you want to know?
    1=Circumference, 2=Surface Area, 3=Volume
    Enter a number from the menu above:
```

The program should accept the user's choice and print the requested information. For reference, the formulas for calculating various characteristics of a sphere are given in Figure 3.10. Use 3.14 as the value of π. If the user enters a number that isn't on the menu, print a friendly error message.

Hint: When calculating the volume, be careful not to write `4/3`. When C sees two integers like this, it assumes you want an integer answer, and gives you a value of 1. Instead, write `4.0/3`. This gives C a clue that you want an answer with decimal places. Test your program by finding the volume of a sphere with a radius of 1.5. It should be about 14.1.

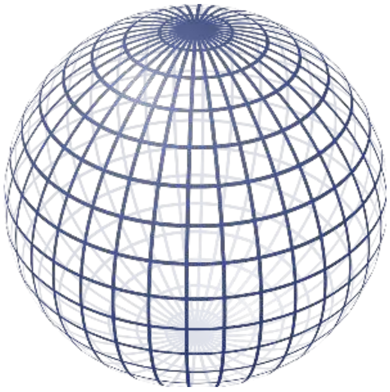


Figure 3.10: Sphere formulas:
Circumference = $2\pi r$
Area = $4\pi r^2$
Volume = $\frac{4}{3}\pi r^3$
Source: Wikimedia Commons

13. Write a program named `lenconv.cpp` that lets the user pick units of feet or meters, then enter a distance in those units. The program should then print a list showing the equivalent distance in other units. The other units should be:

- Feet – Length of Charlemagne's foot
- Meters – Distance travelled by light in 1/299792458th of a second
- Miles – Latin *mille passus*, "a thousand paces"
- Furlongs – The distance a team of oxen can plough without resting
- Fathoms – The width of your outstretched arms
- Smoots – The height of Oliver R. Smoot

Figure 3.11 gives the necessary conversion factors.

When you run the program, it should look like this:

```
This program will convert distances in feet or
meters to other units.
What unit will you start with?
1=feet, 2=meters
1
Enter the distance: 6.5
6.500000 feet is 6.500000 feet
6.500000 feet is 1.981103 meters
6.500000 feet is 0.001231 miles
6.500000 feet is 0.009848 furlongs
6.500000 feet is 1.083333 fathoms
6.500000 feet is 1.164249 smoots
```

meters	=	feet/3.281
miles	=	feet/5280
furlongs	=	feet/660
fathoms	=	feet/6
smoots	=	feet/5.583

feet	=	meters*3.281
miles	=	meters/1609
furlongs	=	meters/201
fathoms	=	meters/1.829
smoots	=	meters/1.702

Figure 3.11: Converting feet and meters to other units. (Picture shows Charlemagne, whose foot originally defined the length of a "foot".)
Source: Wikimedia Commons

14. *Dungeons & Dragons* is a role-playing game that uses dice to determine the outcome of players' actions. Sometimes instead of rolling a single die, the player will be asked to roll two dice and use the larger of the two numbers. This is called "rolling with advantage", since it's advantageous to the player to be able to use the best of the two numbers.

    When rolling a single six-sided die, the average number you'll get is $(1+2+3+4+5+6)/6 = 3.5$. What's the average number you get if you roll two dice and pick the biggest number each time?

    To figure this out, start with Program 2.4 from Chapter 2. Call your new program `advantage.cpp`. Modify the program by adding two new variables, like this:

    ```
    int roll;          // Roll number.
    double sum = 0;    // Sum of numbers rolled.
    ```

    The original Program 2.4 generated one random number, simulating one roll of a six-sided die. Your new program will use a loop like this:

    ```
    for ( roll=0; roll<10000; roll++ ) {
    ```

    to simulate rolling 10,000 times. For each roll, instead of just one random number, generate two random numbers in the range 1 through 6. Then use an "`if/else`" statement to find the bigger of the two numbers and add it to the value of `sum`. At the end of the program, print the average of all the larger numbers, which will just be `sum/10000`.

    Is the result bigger than the value of 3.5 you'd get by rolling just one die? Youtuber and "Stand-up Mathematician" Matt Parker has a good video about this problem here:
    https://www.youtube.com/watch?v=X_DdGRjtwAo

Happy gamers
Source: Wikimedia Commons

15. Some integers are equal to the sum of the squares of two other positive integers. For example, $13 = 2^2 + 3^2$ and $29 = 2^2 + 5^2$. For some numbers, you can even do it in more than one way: $50 = 1^2 + 7^2$ but also $50 = 5^2 + 5^2$. For some numbers, you can't do it at all. There's no way to make 9 out of a sum of two such squares.

    Let's write a program called `squaresum.cpp` that asks the user for an integer, then tries to find pairs of positive integers that can be squared and added to give that number. For example, if we run the program and enter 29 the output should look like this:

```
Enter n: 29
2, 5
```

telling us that $29 = 2^2 + 5^2$. If we enter 50, the program should say this:

```
Enter n: 50
1, 7
5, 5
```

informing us that there are two different pairs of integers whose squares add up to 50. If there's no way to make a number out of a sum of two squares, the program should just print nothing.

If the original number is stored in a variable named `n`, then the program should try to find two numbers, `i` and `j`, so that `i*i + j*j == n`.

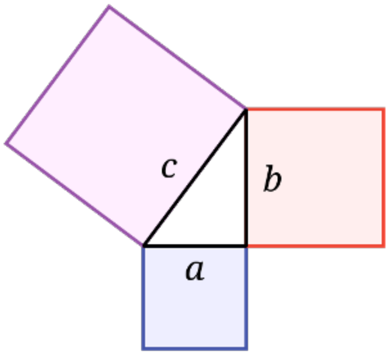
To do this, your program should use a pair of nested loops. You'll need to try all values of `i` from `1` to `n-1`. Then, for each value of `i` you'll need to check each value of `j` between `i` and `n-1` to see if the sum of the squares of `i` and `j` adds up to the right value.

Note that we're only checking values of `j` that are greater than or equal to `i`. Otherwise we would find each pair of numbers twice, like this:

```
Enter n: 29
2, 5
5, 2
```

Here the two answers are the same, they're just in a different order. If your program prints duplicate answers like this, check your "`for`" loops and look for typos.

You might notice that some values of n are themselves perfect squares, like 25, which is 5^2. Your program will tell you that 25 is equal to $3^2 + 4^2$, so $5^2 = 3^2 + 5^2$. When you find a number like this, you've found an integer solution to the equation $c^2 = a^2 + b^2$, which is just the Pythagorean Theorem relationship for a right triangle with a hypoteneuse of length c.



The Pythagorean Theorem states a relationship between the squares of the lengths of the sides of a right triangle.

Source: Wikimedia Commons

16. Some integers are equal to the sum of the cubes of two positive integers. For example, $9 = 1^3 + 2^3$ and $35 = 2^3 + 3^3$. For some numbers, you can even do it in more than one way: $1729 = 1^3 + 12^3$ but also $1729 = 9^3 + 10^3$. For some numbers, you can't do it at all. There's no way to make 10 out of a sum of two such cubes.

    Let's write a program called `cubesum.cpp` that asks the user for an integer, then tries to find pairs of positive integers that can be cubed and added to give that number. For example, if we run the program and enter 9 the output should look like this:

    ```
    Enter n: 9
    1, 2
    ```

    telling us that $9 = 1^3 + 2^3$. If we enter 1729, the program should say this:

    ```
    Enter n: 1729
    1, 12
    9, 10
    ```

    informing us that there are two different pairs of integers whose cubes add up to 1729. If there's no way to make a number out of a sum of two cubes, the program should just print nothing.

    If the original number is stored in a variable named `n`, then the program should try to find two numbers, `i` and `j`, so that `i*i*i + j*j*j == n`.

    To do this, your program should use a pair of nested loops. You'll need to try all values of `i` from `1` to `n-1`. Then, for each value of `i` you'll need to check each value of `j` between `i` and `n-1` to see if the sum of the cubes of `i` and `j` adds up to the right value.

    Note that we're only checking values of `j` that are greater than or equal to `i`. Otherwise we would find each pair of numbers twice, like this:

    ```
    Enter n: 9
    1, 2
    2, 1
    ```

    Here the two answers are the same, they're just in a different order. If your program prints duplicate answers like this, check your "`for`" loops and look for typos.

If you experiment with your program you'll find that it's hard to find numbers that are the sum of cubes. Most numbers will cause your program to print nothing at all. If you look at a lot of numbers, you might notice a more subtle pattern: There are no solutions for any value of n which is itself a perfect cube. This is telling us that there are no integer solutions to the equation $x^3 = y^3 + z^3$. In fact, there are no integer solutions to any equation of the form $x^p = y^p + z^p$ when p is anything bigger than 2. This was postulated by Pierre de Fermat around 1637, but wasn't proven until 1994, by Andrew Wiles.

Pierre de Fermat

Source: Wikimedia Commons

There's also something fun about the number 1729. It's known as Ramanujan's Taxicab Number. Srinivasa Ramanujan and G. H. Hardy were both brilliant mathematicians. One day, Hardy commented to Ramanujan that some numbers were just boring. Take for example, he said, the number on the taxicab I arrived in: 1729. Ramanujan immediately disagreed, and told Hardy that 1729 was actually very interesting, because it was the smallest integer that could be written as the sum of two cubes in two different ways (as we saw above). Ramanujan knew this because he'd been trying to prove Fermat's theorem, and had been looking at lots of sums of cubes.

G. H. Hardy (left) and Srinivasa Ramanujan (right).

Source: Wikimedia Commons, Wikimedia Commons

A nano Cheat-Sheet

Here are a few tips and tricks that will make it easier for you to edit files with *nano*.

- **Delete lines:** To delete a line, move to beginning of line, then press Ctrl-k (hold down the CTRL key, and press the K key). The "K" is for "**K**ut".
- **Cutting and Pasting:** First, move to the beginning of the text you want to cut and press Ctrl-6. Then move to end of the text you want and press Ctrl-K. This will "**K**ut" the text. Now move to the place where you want to insert the text and press Ctrl-U (for "**U**ncut"). Your text will be inserted here. If you want to paste the same text in another location, move there and press Ctrl-U again. You can do this as many times as you want.
- **Search:** To search for text, press Ctrl-W (for "**W**here is..."). You'll be asked what to search for. Enter it, then press the Enter or Return key. The cursor will jump forward to the first occurrance of the text you're searching for. If there are no matches, you'll see a message at the bottom telling you that the thing you searched for wasn't found. To search for the same thing again, press Crtl-W again.
- **Find and Replace:** Press Alt-R (hold down the ALT key and press the R key). You'll be asked what to search for. Enter it, then press the Enter or Return key. You'll be asked for replacement text. Enter this, and press Return again. Finally, you'll be asked whether you want to replace just the first occurrence, or all occurrences.
- **Saving Your Work Without Exiting nano:** To save your work at any time, press Ctrl-O (that's the letter O, not a zero).
- **Displaying the Current Line Number:** *nano* can optionally display the current line number (the number of the line where the cursor currently is). This can be useful when *g++* give you an error message like:

  ```
  hello.cpp:3:27: error: 'prantf' was not declared in this scope
  ```

  In the example above, *g++* is telling us that our program has an error on line 3. (It also tells us that *g++* thinks the error was around character number 27 on that line, but this number is often unreliable.) You can ask nano to temporarily turn on line numbers by pressing Alt-C (meaning "Hold down the Alt key and the C key simultaneously"). This will show line and column numbers near the bottom of *nano's* window.

  If you like, you can make *nano* always display line and column numbers by using *nano* to create a file named /.nanorc and putting the following line into that file:

  ```
  set const
  ```

  The next time you start *nano* it should automatically display the line number at the bottom of the window.

4. *Math and More Loops*

4.1. Introduction

In 1965, Gordon Moore observed that the density of components in integrated circuits (such as computer CPUs) was doubling every year or two[1]. This observation came to be known as "Moore's Law" and it continued to be valid for several decades, although recently the rate has slowed[2]. Similar "Moore's Laws" have been observed for other computer components, such as disk drives, memory, and displays.

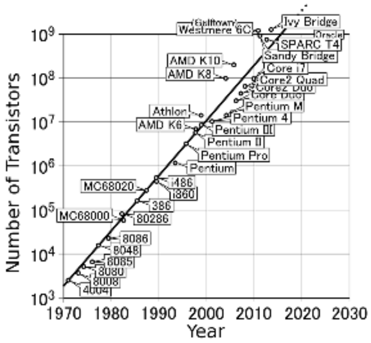


Figure 4.1: An illustration of "Moore's Law" for CPUs. Note that the vertical axis is logarithmic.

Source: *Wikimedia Commons*

As we saw in Chapter 2, modern computers can do thousands of calculations in the blink of an eye. In the final version of our "gutter" program (Program 2.7) we used nested "`for`" loops to simulate the behavior of ten thousand stones during ten rainstorms, and our program ran in less time than it took you to read this sentence.

The first "PC": The *IBM PC 5150*, introduced in 1981.

Source: *Wikimedia Commons*

Computers are very good at doing things over and over again very rapidly. Previously we've used "`for`" loops for this. In this chapter, we'll look at several other kinds of loops available in the C programming language. We'll start out by using a "`for`" loop to test how fast your computer is. Along the way, we'll find out about C's math functions and use them to give your computer something substantial to chew on.

4.2. Math Functions in C

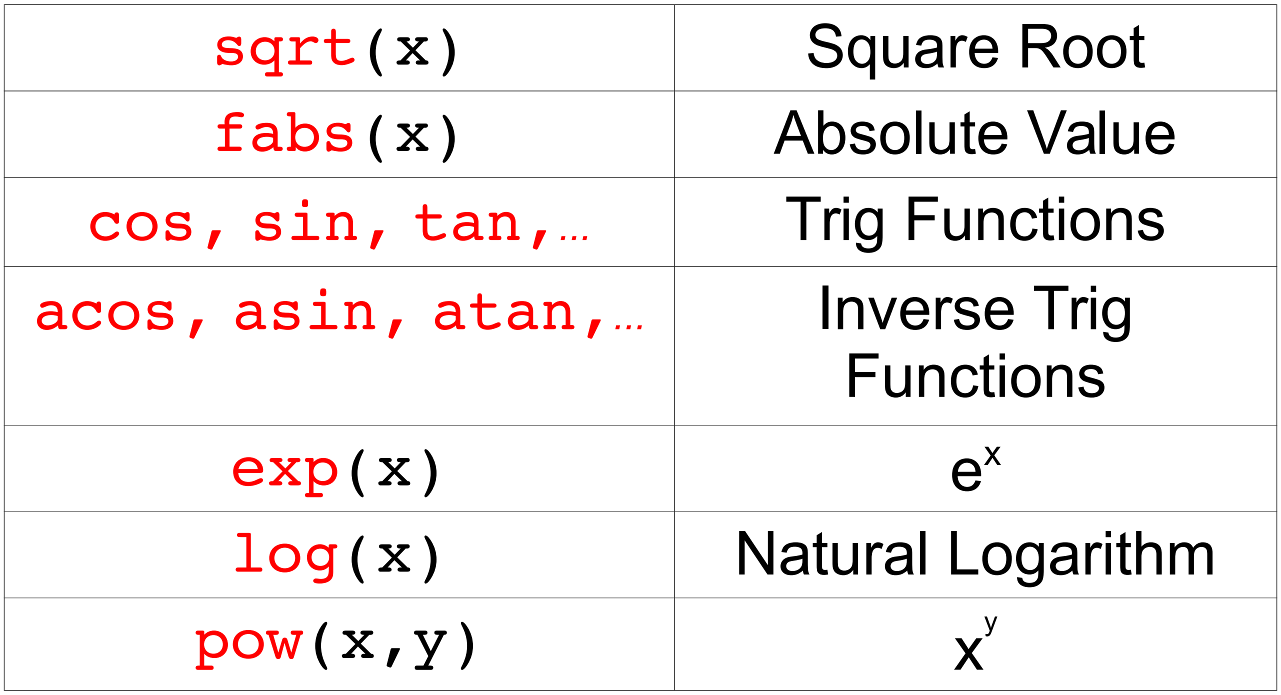
C provides a rich set of math functions and some predefined math constants such as the value of π. Table 4.2 shows some of the most commonly-used functions.

A modern supercomputer: NASA's *Pleiades* Cluster.

Source: *Wikimedia Commons*

[1] Moore, G. E. Electronics 38, 114-117 (1965).

[2] Nature 530, 144-147 (11 February 2016).



sqrt(x)	Square Root
fabs(x)	Absolute Value
cos, sin, tan,…	Trig Functions
acos, asin, atan,…	Inverse Trig Functions
exp(x)	e^x
log(x)	Natural Logarithm
pow(x,y)	x^y

Figure 4.2: Some of C's commonly-used math functions.

As we learned in Chapter 1, functions in C are a lot like the functions you've used in math class. We give the function some number of arguments, and the function gives us back a value. In C the expression `y = cos(x);` means "make the variable `y` equal to the cosine of the value in the variable `x`". We'll learn a lot more about how C functions work in Chapter 9. For now, it's important to know that most of C's math functions require `double` values for their arguments, and these functions also give back a `double` value.

To use these functions in your programs, you'll need to add another "`#include`" statement at the top of your program, like this:

```
#include <math.h>
```

But what about...?

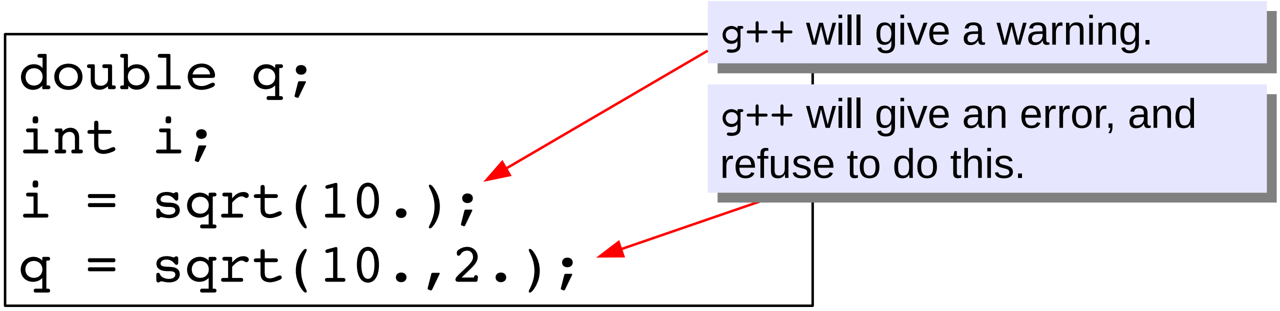
What do these `#include` statements do, anyway? The answer is that they insert chunks of text from other files into your program.

Somewhere on your computer there's a file called `math.h` that contains information about how math functions like `sqrt` are supposed to be used. The information in this file allows *g++* to check that you're using sqrt correctly: Are you giving the function the right number of arguments? Are you putting the value returned by sqrt into the right kind of variable?

For example, sqrt takes one double number as an argument, and it returns a double number. Take a look at Figure 4.3. It shows a couple of incorrect ways to use the sqrt function.

In the first case, the programmer puts the output of sqrt into an integer variable. Since sqrt returns a double number, this means that the decimal part of the number will be chopped off. The *g++* compiler will warn you about this, but it will go ahead and compile the program.

In the second case, the programmer has made a worse mistake. The sqrt function takes only *one* argument, but it's been given *two*. The *g++* compiler doesn't know what the programmer wants it to do, so it emits an error message and refuses to compile the program.



```
double q;
int i;
i = sqrt(10.);
q = sqrt(10.,2.);
```

g++ will give a warning.

g++ will give an error, and refuse to do this.

Figure 4.3: Wrong ways to use the sqrt function.

The math.h file also defines values for some common constants. For example, if you need the value of π in your program, you can just write M_PI, and for the base of natural logarithms (e), you can write M_E.

4.3. How Fast is Your Computer?

Let's use one of these math functions to test how fast your computer is. Take a look at Program 4.1. This program uses the sqrt function, and sums up the square roots of a *billion numbers*!

Source: Wikimedia Commons

The program uses C's "exponential notation", which makes it easier to write large numbers. Instead of writing `1000000000` we can write `1e+9`, meaning "$1x10^9$". Here are some more examples:

$$\begin{aligned}
\texttt{2.5e+3} &= 2,500 \\
\texttt{6.02e+23} &= 6.02 \times 10^{23} \text{ ($\simeq$ Avogadro's number)} \\
\texttt{5e-11} &= 5 \times 10^{-11}
\end{aligned}$$

Notice that 10^3 is just `1e+3` ("one times ten to the third power"), *not* `10e+3`. Here the `e` means "times ten to the ...".

Program 4.1 begins by recording the current time[3] in the variable `tstart`. After summing up all of the numbers, the program looks at the new time, and prints out how long, in seconds, the program ran.

Notice that the `sqrt` function, like all of the math functions we'll be using, takes `double` arguments and returns a `double` value. Because the variable `i` is an integer, we need to "cast" it as a double by saying `(double)` in front of it. If we didn't do this *g*++ would complain.

Why do we set `sum` equal to zero before we start the program's loop? Won't it just be zero automatically? No, not necessarily. You shouldn't assume that a variable has any particular value before you explicitly give it one. Remember that variables are temporary boxes in the computer's memory. After the program is done with them, the same chunk of memory can be re-used by other programs. In some cases, if you don't explicitly give a variable a value, it will contain whatever random data happens to be at that memory location, leftover from the last program that used it.[4]

Program 4.1: timer.cpp (Version 1)

```
#include <stdio.h>
#include <time.h>
#include <math.h>
int main () {
  int i;
  int tstart;
  int delay;
  double sum = 0.0;

  tstart = time(NULL);

  for ( i=0; i<1e+9; i++ ) {
    sum = sum + sqrt( (double)i );
  }

  delay = time(NULL) - tstart;
  printf ("Sum is %lf\n", sum );
  printf ("Total time %d sec.\n", delay );
}
```

[3] in terms of the number of seconds since January 1, 1970. You might remember the `time` function from Chapter 2, where we used it to pick a "seed" for our pseudo-random number generator.

[4] Some compilers will automatically set all variables to zero at the beginning of a program, but it's best not to assume this.

This is important for a variable like sum in Program 4.1. Notice the line in bold. Each time around the loop, this sets the new value of *sum* equal to the old value plus $\sqrt{i}$. If we didn't explicitly set sum = 0.0 before we began adding things up, then the "old value" of sum would be undefined (and possibly some bizarre, unexpected number) the first time we went through the loop.

Exercise 21: How Fast is Your Computer?

Create, compile and run Program 4.1. On a typical computer, it should take no more than a minute or two to run. If you find that it takes longer, press Ctrl-C to stop it, and try reducing the number of loops by a factor of ten. How many square roots per second can your computer do?

Another kind of progress: A Russian *Progress* cargo spacecraft departing from the International Space Station. The computers that control the ISS aren't particularly new or fast. They're tried-and-true technology chosen for its reliability. The "Vehicle Management Computers", for example, are many redundant computers each powered by an Intel 386SX CPU running at 32 MHz. This is 100 times slower than the CPUs in most modern laptop and desktop computers.

Source: *Wikimedia Commons*

4.4. Progress Reports

While your timer program was running, you may have worried that it wasn't actually doing anything. It's often useful to make your program print out reports periodically, so you can see its progress. Let's modify Program 4.1 and make it do this. We'll use a new mathematical operator to help us.

The "modulo" (or "modulus") operator, "%", does one peculiar but useful thing: it tells us the remainder left over after we do division. For example, "10 % 5" would be equal to zero, since the remainder after dividing ten by five is zero. Here are some other examples:

10 % 7	gives	3
1001 % 10	gives	1
25 % 7	gives	4

Program 4.2 uses the modulo operator to print out the elapsed time, and the number of square roots that have been summed so far, every million times around the loop. It does this by looking at i % 1000000 (we can read this as "i modulo one million"). When this quantity is zero, it means that i is a multiple of 1,000,000.

Exercise 22: Speed Test with Progress Report

Create, compile, and run Program 4.2. Does it behave as expected? Is it more entertaining to see evidence that the program is doing something?

Program 4.2: timer.cpp (Version 2)

```
#include <stdio.h>
#include <time.h>
#include <math.h>
int main () {
  int i;
  int tstart;
  int delay;
  double sum = 0.0;

  tstart = time(NULL);

  for ( i=0; i<1e+9; i++ ) {
    sum = sum + sqrt( (double)i );
       if ( i%1000000 == 0 ) {
          delay = time(NULL) - tstart;
          printf ("Time after %d terms: %d sec.\n", i, delay );
        }
  }

  delay = time(NULL) - tstart;
  printf ("Sum is %lf\n", sum );
  printf ("Total time %d sec.\n", delay );
}
```

But what about...?

What does "modulo" mean anyway? Where does that word come from?

Take a look at the two clocks in Figure 4.4. Can you tell how much time has passed? Not necessarily, because clocks count to twelve, and then they start over again. This is what mathematicians call "modular arithmetic". In the case of the clocks, we could say that they have a "modulus" of twelve.

For example, if we start at midnight and wait 28 hours, the little hand on the clock will be pointing to `28 % 12` ("28 modulo 12"), which is 4.

In modular arithmetic, two numbers that have the same remainder when divided by the modulus are said to be "congruent". A mathematician would say that 2 AM and 2 PM are congruent in the clock's modular arithmetic.

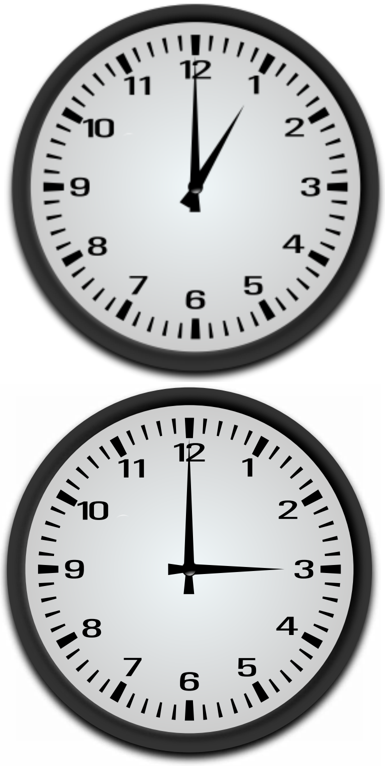
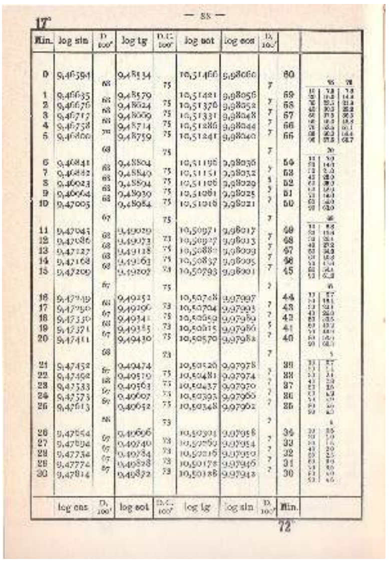


Figure 4.4: Have two hours passed, or 14 hours? Or even a 26 hours? We can't tell. Source: Openclipart.org

4.5. Trigonometric Functions

The advantages you young people have! Take a look at Figure 4.5. Back in the days before pocket calculators, if your ancestors needed to find the sine or cosine of an angle they looked up the values in "trig tables" like this one. Think about the hours of work that went into constructing these tables. The numbers had to be computed by hand, using tedious mathematical techniques to find the value of each function at given angles. One of the first tasks given to early computers like ENIAC (1945-1947, Figure 4.6) was the creation of mathematical tables, particularly those needed for aiming artillery shells.



17° — 88 —

Min.	log sin	D. 100"	log tg	D.C. 100"	log cot	log cos	D. 100"	
0	9,46594		9,48534		10,51466	9,98060		60
		68		75			7	
1	9,46635		9,48579		10,51421	9,98056		59
		68		75			7	
2	9,46676		9,48624		10,51376	9,98052		58
		68		75			7	
3	9,46717		9,48669		10,51331	9,98048		57
		68		75			7	
4	9,46758		9,48714		10,51286	9,98044		56
		70		75			7	
5	9,46800		9,48759		10,51241	9,98040		55
		68		75			7	
6	9,46841		9,48804		10,51196	9,98036		54
		68		75			7	
7	9,46882		9,48849		10,51151	9,98032		53
		68		75			5	
8	9,46923		9,48894		10,51106	9,98029		52
		68		75			7	
9	9,46964		9,48939		10,51061	9,98025		51
		68		75			7	
10	9,47005		9,48984		10,51016	9,98021		50
		67		75			7	
11	9,47045		9,49029		10,50971	9,98017		49
		68		73			7	
12	9,47086		9,49073		10,50927	9,98013		48
		68		75			7	
13	9,47127		9,49118		10,50882	9,98009		47
		68		75			7	
14	9,47168		9,49163		10,50837	9,98005		46
		68		73			7	
15	9,47209		9,49207		10,50793	9,98001		45
		67		75			7	
16	9,47249		9,49252		10,50748	9,97997		44
		68		73			7	
17	9,47290		9,49296		10,50704	9,97993		43
		67		75			7	
18	9,47330		9,49341		10,50659	9,97989		42
		68		73			5	
19	9,47371		9,49385		10,50615	9,97986		41
		67		75			7	
20	9,47411		9,49430		10,50570	9,97982		40
		68		73			7	
21	9,47452		9,49474		10,50526	9,97978		39
		67		75			7	
22	9,47492		9,49519		10,50481	9,97974		38
		68		73			7	
23	9,47533		9,49563		10,50437	9,97970		37
		67		73			7	
24	9,47573		9,49607		10,50393	9,97966		36
		67		75			7	
25	9,47613		9,49652		10,50348	9,97962		35
		68		73			7	
26	9,47654		9,49696		10,50304	9,97958		34
		67		73			7	
27	9,47694		9,49740		10,50260	9,97954		33
		67		73			7	
28	9,47734		9,49784		10,50216	9,97950		32
		67		73			7	
29	9,47774		9,49828		10,50172	9,97946		31
		67		73			7	
30	9,47814		9,49872		10,50128	9,97942		30
	log cos	D. 100"	log cot	D.C. 100"	log tg	log sin	D. 100"	Min.

72°

Figure 4.5: Math tables were once widely used to find values for trigonometric functions, logarithms, and other functions. Source: Wikimedia Commons

For a good overview of the techniques used in constructing such tables, see this Wikipedia article

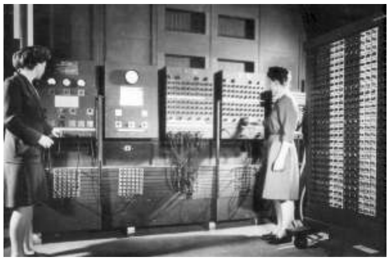


Figure 4.6: Betty Jennings and Frances Bilas operating ENIAC.

Source: Wikimedia Commons

Modern computers make this much easier for us. Let's write a program that uses C's math functions to generate a table of values for $cos(\theta)$ and $sin(\theta)$ for various values of θ. Before we start, it might be good to remind ourselves what sine and cosine are. Take a look at Figure 4.7. If you imagine a point travelling along the circumference of a circle with a radius of 1, then $cos(\theta)$ and $sin(\theta)$ are just the x and y coordinates of the point when it's at the angle θ. Let's start out with $\theta = 0$ and move around the circle in 100 steps, until we get back to where we started.

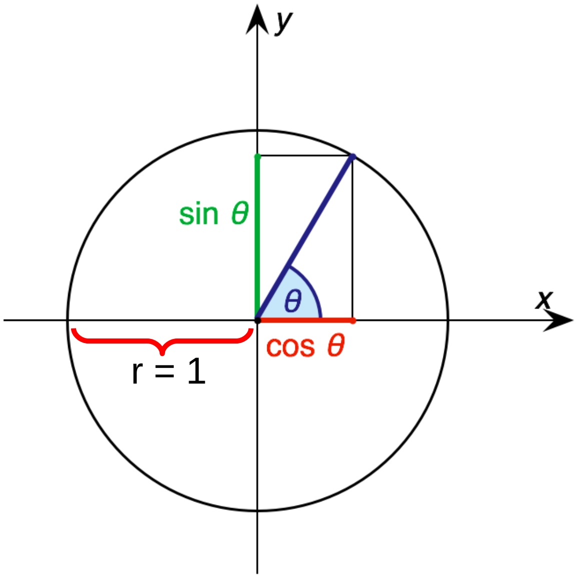


Figure 4.7: The definition of sine and cosine.

Source: Wikimedia Commons

Remember that there are two different systems for measuring angles: degrees and radians. When you go all the way around a circle, you've turned by 360°. This is equivalent to 2π radians. C's trigonometric functions all use **radians**, so our program will need to divide 2π radians into 100 steps, and calculate the sine and cosine for each.

That's what we do in Program 4.3. Notice that we're careful to set

Program 4.3: trig.cpp

```
#include <stdio.h>
#include <math.h>
int main () {
  double theta = 0.0;
  double step = 2.0 * M_PI / 100.0;
  int i;

  for ( i=0; i<100; i++ ) {
    printf ( "%lf %lf %lf\n", theta, cos(theta), sin(theta) );
    theta += step;
  }

}
```

theta equal to zero at the beginning, just as we did with sum in Program 4.1. Each time around the loop, we add a little bit to theta until we've worked our way completely around the circle. The size of each step is $2\pi/100$, since the whole circle is 2π radians and we want to divide it up into 100 steps.

Also notice that we use the symbol M_PI that's conveniently provided for us by math.h.

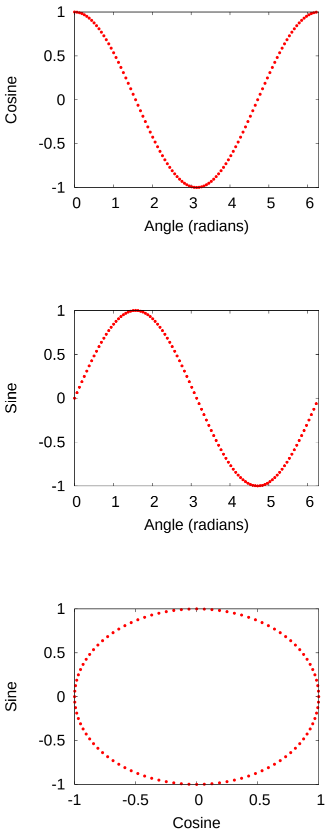


Figure 4.8: Plots of θ versus $cos(\theta)$, θ versus $sin(\theta)$, and $cos(\theta)$ versus $sin(\theta)$.

Exercise 23: Making a Trig Table

Create, compile, and run Program 4.3. It should make three columns of text, containing values for θ, $cos(\theta)$ and $sin(\theta)$. Now run it again, like this, to write the table into a file:

```
./trig > trig.dat
```

It's hard to see whether your program is doing the right thing by just looking at the numbers. Let's try graphing them. Start up *gnuplot* by typing its name, and then give it this command:

```
plot "trig.dat"
```

You should see something that looks like the top graph in Figure 4.8. Now try giving *gnuplot* this command:

```
plot "trig.dat" using 1:3
```

You should see something like the middle graph in Figure 4.8. Next try this *gnuplot* command:

```
plot "trig.dat" using 1:2, "trig.dat" using 1:3
```

The result should be the first two graphs laid on top of each other. Finally, try this:

```
plot "trig.dat" using 2:3
```

You should see something like the bottom graph in Figure 4.8.

Hipparchus of Nicea (180-125 BCE) is credited with creating the first trigonometric tables. He's the bearded gentleman shown holding the blue celestial sphere in this detail from *The School of Athens*, by Raffaello Sanzio (1509). Source: Wikimedia Commons

What did *gnuplot* do? The first command told *gnuplot* to plot the contents of the file `trig.dat`, but how did it know which columns to use? The file contains three columns of data: θ, $cos(\theta)$, and $sin(\theta)$. As it turns out, *gnuplot* assumes that the first two columns in a file represent the x and y coordinates of a set of points to be plotted. If the file only contains one column, *gnuplot* uses the line number as x, and the value on each line as y.

Before computers and calculators became widely available, the slide rule was widely used for calculations involving logarithms or trigonometric functions.
Source: Wikimedia Commons

If your file contains more than two columns, you can tell *gnuplot* which ones to use as x and y with the "`using`" qualifier. If you say "`using 1:3`", that means "column 1 is x and column 3 is y". We can ask *gnuplot* to superimpose multiple graphs by giving it a comma-separated list of things to plot, as we did in the next-to-last "`plot`" command in the exercise above.

As you can see from the bottom graph in Figure 4.8, our values for $cos(\theta)$ and $sin(\theta)$ really do correspond to the x and y values of a point at various angles, as they should. (The circle looks flattened because the vertical and horizontal scales are different. By default, *gnuplot* fits its graphs into a rectangular window that's wider than it is tall. You can fix this by telling gnuplot "`set size square`".)

4.6. Using "`while`" Loops

Until now we've used just one of the kinds of loops that the C programming language provides. The "`for`" loop that we've been using is what programmers call a "counted" loop, because we tell the computer how many times to go around the loop. Another kind of loop is called a "conditional" loop. We can create one of these using C's "`while`" statement, which looks like this:

```
while (CONDITION) {
   BLOCK OF STATEMENTS
}
```

The statements inside the loop will be acted upon again and again, as long as the "CONDITION" is true. You might notice that this looks a lot like an "`if`" statement, where a block of statements is only executed if some condition is met. With "`if`", the block of statements is only acted upon once, but with `while` they're done over and over, for as long as the condition continues to be met. Here's an example:

```
int i = 0;
while ( i < 10 ) {
        printf ( "%d\n", i );
        i++;
}
```

The code in this example would print out the integers from zero to nine. This is the same kind of thing we've done with "`for`" loops, but done in a different way. Consider the following example, though:

```
int i = 0;
while ( i < 1000000 ) {
        i = rand();
        printf ( "%d\n", i );
}
```

The second example will continue printing random numbers until it finds one that's greater than 1,000,000, and then it will stop. We don't know in advance how many times the computer will go around the loop. The number of loops just depends on the condition we set in the `while` statement. That's why this kind of loop is called a "conditional" loop.

How many loops are in this roller coaster?

Source: Wikimedia Commons

4.7. Writing a Game

Program 4.4 also uses a `while` loop. In this case, we're playing a game like Blackjack. Blackjack (also know as Twenty-One) is a card game where each player is dealt cards, one card at a time. Each card has a numerical value from one to thirteen. The object of the game is to get the sum of all your cards as close to twenty-one as possible, without going over. Each time Program 4.4 goes through its `while` loop, it picks a random number from one to thirteen, then adds this number to the sum so far. It keeps doing this for as long as the sum is less than twenty-one.

Traditional playing cards have either numbers or faces on them. The values of the numbers are self-explanatory. For the faces, we count Jack, Queen and King as 11, 12 and 13, respectively.

Source: Wikimedia Commons

Program 4.4: addem.cpp (Version 1)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
  int sum = 0;
  int card;

  srand(time(NULL));
  while ( sum < 21 ) {
    card = (int)( 1 + 13.0*rand()/(1.0 + RAND_MAX) );
    sum += card;
    printf ("Got %d. Sum is now %d\n", card, sum );
  }
}
```

Do you see how this makes a number between 1 and 13?

Exercise 24: Add 'Em Up!

Create, compile and run Program 4.4. Does it work as expected? Run it several times to see if you can hit exactly twenty-one.

The card-player, by Aba Novak.

Source: Wikimedia Commons

We could improve on Program 4.4 by telling it to congratulate us when we win. To do this we might modify the `while` loop to make it look like this:

```
  while ( sum < 21 ) {
    card = (int)( 1 + 13.0*rand()/(1,0 + RAND_MAX) );
    sum += card;
    printf ("Got %d. Sum is now %d\n", card, sum );
    if ( sum == 21 ) {
      printf ("You WIN!\n");
    }
  }
```

4.8. Stopping or Short-Circuiting Loops

The problem with our game so far is that there's no skill involved in playing it. It's purely random whether you win or lose.

In the real game of Blackjack, after each card is dealt the player is asked whether he/she wants another. If the player is very close to twenty-one already, he or she may choose not to get any more cards, hoping that all of the other players will either go over twenty-one, or not get as close. (Whichever player gets closest to twenty-one, without going over, wins.) Let's modify our program to allow for this. Take a look at Program 4.5.

The Card Players by Catherine Ann Dorset. (Note that one of the players seems to be a Great Auk, which sadly became extinct in the mid nineteenth Century.)

Source: Wikimedia Commons

Program 4.5: addem.cpp (Version 2)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
  int sum = 0;
  int card;
  int ans;

  srand(time(NULL));
  while ( sum < 21 ) {
    card = (int)( 1 + 13.0*rand()/(1.0 + RAND_MAX) );
    sum += card;
    printf ("Got %d. Sum is now %d\n", card, sum );

    if ( sum == 21 ) {
      printf ("You WIN!\n");
    } else if ( sum > 21 ) {
      printf ("You lose!\n");
    } else {
      printf ("Enter 1 to continue or 0 to quit while you're ahead: ");
      scanf("%d", &ans);
      if ( ans != 1 ) {
        printf("Your final score was %d\n",sum);
        break;
      }
    }
  }
}
```

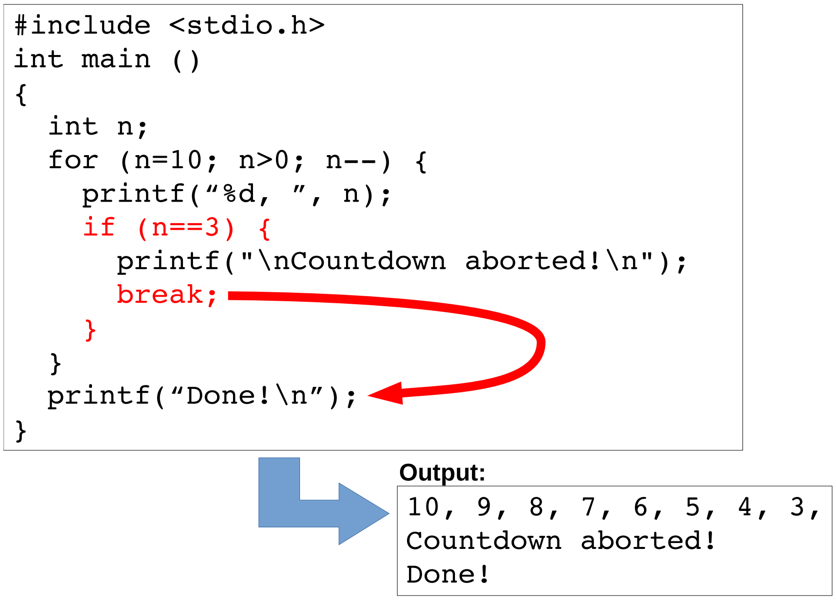
As you can see, we've added an "`if`" statement to deal with the various possible outcomes. If the sum is exactly twenty-one, we tell the player he or she has won. If it's over twenty-one, we identify the player as a loser. If the sum is under twenty-one, we give the player a choice: continue or quit? If the player chooses to continue, we go around the loop again.

But what if the player chooses to quit? How can we make the loop stop right now, without waiting for the sum to get greater than twenty-one? To do this, we use the C language's "`break`" statement. A `break` statement causes the loop it's in to stop immediately.

Exercise 25: Playing a Card Game

Create, compile and run Program 4.5. Try running it several times, making sure you sometimes tell it to continue, and sometimes tell it to quit. Does it behave as expected?

Figure 4.9 shows another program that uses the `break` statement. The program in the figure does a countdown, from ten toward zero, but before it reaches zero the countdown is stopped by using `break`.



```
#include <stdio.h>
int main ()
{
  int n;
  for (n=10; n>0; n--) {
    printf(“%d, ”, n);
    if (n==3) {
      printf("\nCountdown aborted!\n");
      break;
    }
  }
  printf(“Done!\n”);
}
```

Output:

```
10, 9, 8, 7, 6, 5, 4, 3,
Countdown aborted!
Done!
```

Figure 4.9: Using `break` to stop a loop.

C's `break` statements are often useful when your program is searching for something. Imagine you're looking through a big stack of books, trying to find one with a particular title. You start from the top and look at the books one at a time until you find the one you want. Then you stop. You don't keep looking through the rest of the stack.

You can use `break` to do something similar in a C program. When we find the thing we're looking for, we can immediately stop looping and go on with the rest of the program.

But what about. . . ?

What if you use break inside two or more nested loops, like this?:

```
for ( i=0; i<nrocks; i++ ) {
    for ( j=0; j<nstorms; j++ ) {
        ...
        break;
    }
}
```

This is similar to the nested loops in Program 2.7, which tracked each of many rocks as they were washed down a gutter by some number of rainstorms.

The break statement only halts the innermost loop containing it. In the example above, the break would stop the nstorms loop, and the computer would go back to the top of the nrocks loop. If there were more rocks left to do, it would continue with the next rock, and start the nstorms loop again for the new rock.

Compare that with the following example:

```
for ( i=0; i<nrocks; i++ ) {
    for ( j=0; j<nstorms; j++ ) {
        ...
    }
    ...
    break;
}
```

In the second example, the break statement would stop the outer, nrocks, loop, and the computer would continue without doing anything else with either of these loops.

What if you wanted to skip the rest of this trip around a loop, but not stop looping? You can do that, too, using C's "continue" statement.

Consider the following example:

```
for ( i=0; i<10; i++ ) {
    printf ("Loop number %d\n", i);
    if ( i >= 5 ) {
        continue;
    }
    printf ("This number is below 5.\n");
}
```

If we ran a program containing this code, it would print:

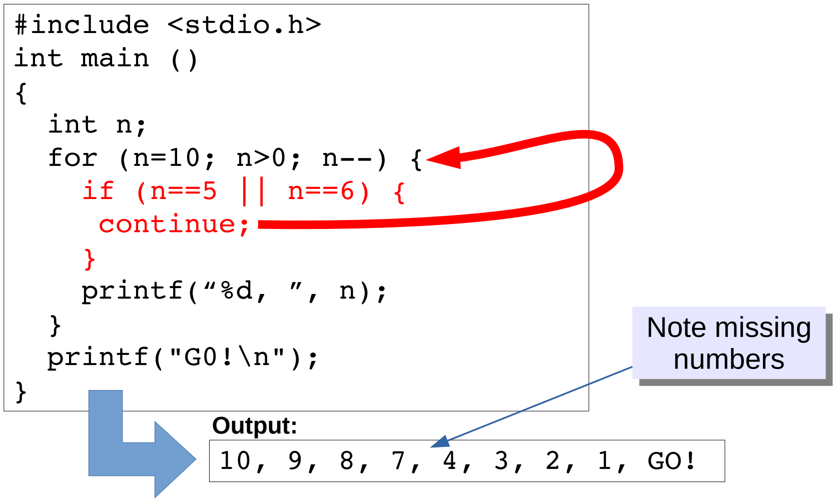
```
Loop number 0
This number is below 5.
Loop number 1
This number is below 5.
Loop number 2
This number is below 5.
Loop number 3
This number is below 5.
Loop number 4
This number is below 5.
Loop number 5
Loop number 6
Loop number 7
Loop number 8
Loop number 9
```

When the `continue` statement is acted upon, the computer skips everything else in this trip around the loop and goes directly back to the top, to start the next trip. Just like `break`, `continue` only affects the innermost loop containing it.

Figure 4.10 shows another countdown example. This time, for some reason, Mission Control has decided to omit some numbers from the countdown. (Maybe they're superstitious?)

As with the other countdown example, we can imagine an analogy between this and searching for something in the real world. Imagine that you have a stack of books, some of which are paperback and some of which are hardback. You're looking for a particular title, and you remember that it's a hardback book. You'll go through the stack quickly, discarding the paperbacks without even looking at them, and proceeding down the stack.

We can use a `continue` statement to do this kind of thing in a loop. The `continue` causes the current trip around the loop to stop, and the computer goes immediately back up to the top of the loop and starts the next trip.



```
#include <stdio.h>
int main ()
{
  int n;
  for (n=10; n>0; n--) {
    if (n==5 || n==6) {
     continue;
    }
    printf(“%d, ”, n);
  }
  printf("GO!\n");
}
```

Output:

```
10, 9, 8, 7, 4, 3, 2, 1, GO!
```

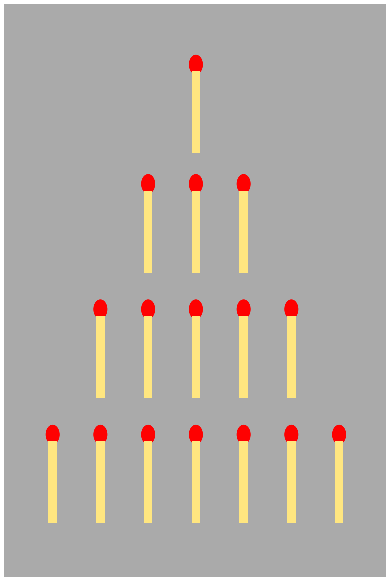
Figure 4.10: Using “continue” to short-circuit a loop.

4.9. Writing a Two-Player Game

Let's use use our new knowledge of `while` loops to write another game. This time, we'll write a two-player game in which the user plays against the computer. It will be a version of an ancient game called "Nim".

In this version of Nim, twelve coins are placed on a table, as in Figure 4.11. The players take turns picking up 1, 2, or 3 coins at a time (the player is free to choose how many coins to take). The player who picks up the last coin wins.

Program 4.6 plays this game. It starts out with 12 coins on the table by setting the variable `coins` equal to 12. After telling the user the rules (using some `printf` statements) the program begins a `while` loop. Each time around the loop one of the players (user or computer) takes some number of coins, and this number is subtracted from `coins`. The `while` loop keeps going as long as the value of `coins` is greater than zero.



There are more complicated versions of Nim. Often it's played by laying out a pyramid of objects (such as the matchsticks shown here), and only allowing players to remove objects from a single row during each turn.

Source: Wikimedia Commons

Figure 4.11: Are you ready for a game of "12-coin Nim"?

Source: Wikimedia Commons (1, 2, 3)

If you try playing this game, you'll find that the computer always wins! By employing a simple strategy, the computer can always win the game. Can you understand how it works?[5]

Notice that the program uses a `continue` statement to keep users from cheating. If the user picks a number other than 1, 2, or 3, the program sends the user back to the top of the loop to try again.

[5] There's an excellent Wikipedia article about the game of Nim and the mathematics behind it. You'll also be amused by Matt Parker's explanation of the game on his YouTube channel, "Standup Maths". Take a look if you can't figure out how the computer's strategy works.

Also notice how the program switches between "Player 0" and "Player 1". After each player's turn, the variable `nextplayer` is set to a value that indicates who the next player should be.

Program 4.6: nim.cpp

```
#include <stdio.h>
int main () {
  int coins = 12;
  int take;
  int nextplayer = 0;  // Player 0=user, 1=computer
  int currentplayer;

  printf ("There are %d coins.\n", coins);
  printf ("You may take 1, 2, or 3 of them.\n");
  printf ("Whoever gets the last coin wins.\n");
  printf ("You are player 0, the computer is player 1.\n");

  while ( coins > 0 ) {

    currentplayer = nextplayer;
    printf ("-------- Player %d's Turn --------\n", currentplayer);

    if ( currentplayer == 0 ) {
       printf ("How many coins will you take?: ");
      scanf("%d", &take);
      if ( take > 3 || take < 1 ) {
        printf ("You must take 1, 2, or 3. Try again\n");
        continue;
      }
       nextplayer = 1;
    } else {
        take = 4 - take;
      printf ("I will take %d of them.\n", take );
       nextplayer = 0;
    }

    coins = coins - take;
    printf ("There are now %d coins left.\n", coins );

  }

  printf("Player %d Wins!\n", currentplayer);
}
```

Keep looping until all coins are gone

Player 0

The computer's winning strategy

Player 1

4.10. One More Kind of Loop

Programmers say that `for` loops and `while` loops are both “pre-test loops”. Take a look at the partial program below, containing a `for` loop and a `while` loop:

```
int nloops = 0;
int i;

for ( i=0; i<nloops; i++ ) {
    printf ( "%d\n", i );
}

while ( nloops > 0 ) {
    printf ( "%d\n", i );
}
```

Neither of these loops will print out anything, because their conditions are never satisfied. In the first loop, `nloops` is zero, and `i` will never be less than zero, and the second loop does nothing for a similar reason. The statements in these loops will never be acted upon, not even once.

The C language offers a third kind of loop that's a “post-test loop”. This is the “`do`” loop (also known as the “`do-while`” loop). Consider this example:

```
do {
    printf ( "%d\n", i );
} while ( i < 0 );
```

If we ran the example above, it would always print out something, no matter what the value of `i` is. The statements inside a `do-while` loop will always be acted upon at least once. After each trip through the loop, the `do-while` statement's condition is examined to see whether it's satisfied, determining whether to go around the loop again. A `do-while` loop is sort of an upside-down `while` loop.

The important difference is that statements inside a `do-while` loop will always be acted upon at least once, but there's no guarantee that statements inside a `while` loop will ever be acted upon. `do-while` loops can be useful in cases where initial values are undetermined before the loop starts.

The general form of a do-while loop is this:

```
do {
  BLOCK OF STATEMENTS
} while (CONDITION);
```

4.11. Estimating the Value of π

A π pie. Source: Wikimedia Commons

Take a look at Program 4.7. This program estimates the value of π by using an approximation discovered in the 14th-Century by Indian mathematician Madhava of Sangamagrama. He found that π was given by the sum of the terms of an infinite series:

$$\pi = \sqrt{12}\left(1 - \frac{1}{3 \cdot 3} + \frac{1}{5 \cdot 3^2} - \frac{1}{7 \cdot 3^3} + \cdots\right)$$

Notice that the size of term number n inside the parentheses is:

$$\frac{1}{(1+2n) \cdot 3^n}$$

and that the sign of the terms bounces back and forth between positive and negative. The terms get smaller and smaller as the series goes on.

Program 4.7 starts calculating the terms in this series and adding them up. It keeps going until it comes to a term that's smaller than 10^{-11} (we chose this value arbitrarily, deciding that we could ignore corrections smaller than that). The program uses a do-while loop to do the work. Notice that we use C's pow function to get the value of 3^n when calculating each term, and the fabs function to find the absolute value of the term.[6] The alternating signs of the terms is taken care of by the multiplier variable, which alternates between 1 and -1 (can you see why?).

After each trip through the loop, the computer checks the absolute value (since the terms alternate between positive and negative) of the current term to see if it's less than our cutoff value of 10^{-11}. A do-while loop

[6] See Figure 4.2.

is more convenient than a `while` loop in this case, since we don't know what the value of the first term will be until we've gone through the loop once.

At the end of the program, we print out our estimate of π and compare it to the "actual" value as given by `M_PI`. Notice that we have to multiply our sum by $\sqrt{12}$ to get π (see Madhava's series, above). The program's output looks like this:

```
Pi     = 3.141592653595635 after 21 terms.
Actual = 3.141592653589793
```

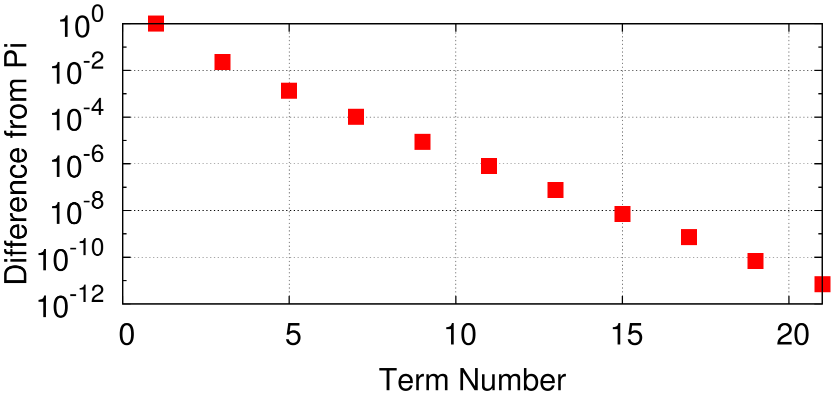


Figure 4.12: The difference betwen our estimate of π and the actual value, as we add more terms to the sum. Note that the vertical scale is logarithmic.

Program 4.7: findpi.cpp

```
#include <stdio.h>
#include <math.h>
int main () {
  double sum = 0.0;
  double term;
  double multiplier = 1.0;
  double small = 1.0e-11;
  int nterms = 0;

  do {
    term =  multiplier / (( 1.0 + 2.0*nterms ) * pow(3.0,nterms));
    sum += term;
    nterms++;
    multiplier = -multiplier;
  } while ( fabs(term) >= small );

  printf ("Pi     = %.15lf after %d terms.\n", sum*sqrt(12.0), nterms );
  printf ("Actual = %.15lf\n", M_PI);
}
```

4.12. Conclusion

C provides a rich set of math functions and a versatile toolkit of loop structures. Together, these allow us to write computer programs that accomplish in seconds tasks that once took many hours of human labor.

To summarize some of the things we've talked about in this chapter:

- To use C's math functions, you need to add `#include <math.h>` to the top of your program.
- The math functions take arguments of type `double`, and return `double` values.
- Several constants are defined in `math.h`, including `M_PI` and `M_E`.

- "`for`" loops are good for situations where you know in advance how many times you want to go around the loop.
- `while` loops are good when you want to keep going until some condition is met.
- `do-while` loops are good when you want to do a test after going through the loop the first time.

Practice Problems

1. Create a modified version of Program 4.1 (the first version of the `timer.cpp` program) that tells you how many square roots per second your computer can do. Call the new program `speedtest.cpp`.

2. Write a program named `clocktime.cpp` that uses **only** addition and just one modulo operator (see the example in Program 4.2) to calculate what number the hour hand of a clock would be pointing to after a given number of hours have passed. The program should ask the user for the current hour, and then ask how many hours in the future. For example, if the user says that the hour is currently 3, and wants to know what the hour will be after 15 hours have passed, the program should say "6". **Hint:** It's OK if your program prints zero when the answer should really be 12.

3. Write a new program called `square.cpp`. The new program should be like Program 4.3, except that:

   (a) instead of θ, $sin(\theta)$ and $cos(\theta)$, the new program should print out two columns: θ and $\sqrt{\theta}$

   (b) instead of going from zero to 2π, do it for 100 steps between zero and ten.

4. Like trig tables, tables of logarithms were also very important to scientists and engineers before calculators and computers were available[7]. One of the first tasks assigned to early computers was the generation of these tables. Write a program named `log.cpp` that uses a `while` loop to generate a list of numbers from 1 to 10, in steps of 0.01, along with the natural logarithm of each number, as given by C's `log` function (see Figure 4.2). Make the program write two columns, separated by a space: The first column should be the number, and the second column should be its log.

   Hints: Define two `double` variables, `x` and `deltax`. Set `deltax = 0.01` and initially set `x = 1`. Then use a `while` loop to print `x` and `log(x)`. Then, before going around the loop again, add `deltax` to `x`. Make the loop stop when `x` is no longer less than ten.

5. Imagine that a very generous bank offers you a nominal annual interest rate of 100% on your investments. If you deposit $1,000 at the beginning of the year and the bank adds 100% at the end of the year, you'd end up with $2,000! Sweet!

   But what if, instead of adding all the interest at the end of the year, the bank gave you 50% interest after six months and another 50%

Portrait of Jacob Bernoulli (1654-1705).
Source: Wikimedia Commons

[7] This Numberphile video by Roger Browley shows how log tables were used: https://www.youtube.com/watch?v=VRzH4xBoGdM.

after another six months? (A banker would say that the interest was "compounded" two times per year.) In the middle of the year you'd have $1,500. Adding another 50% to that at the end of the year would give you a total of $2,250. Even better! And if the bank paid us 25% four times per year we'd end up with $2,441, an even larger amount. Compounding the interest more often apparently gives us more money at the end of the year.

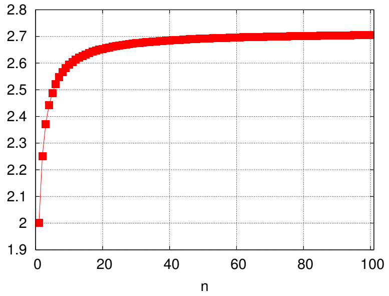


Figure 4.13: This is what a graph of your `interest.cpp` program's output should look like. Notice that the value rises rapidly at first, then levels of to a value approaching e.

In the 17th Century, Jacob Bernoulli realized that you can find out how much money you'll have at the end of the year by multiplying your original investment by:

$$(1 + \frac{1}{n})^n$$

where n is the number of times per year that the interest is compounded. He discovered that there's a limit to how much money you can make, even if you let n go to infinity. In this limit, the expression above approaches a value of about 2.718. Today we know this number as Euler's Constant, e, the base of natural logarithms[8]. So, the most we'd have at the end of the year would be about $2,718, no matter how often the interest is compounded.

Write a program named `interest.cpp` that uses the `pow` function (see Figure 4.2) to evaluate the mathematical expression above. For each value of n from 1 to 100 print n and the expression's value. (The program's output should be two columns of numbers.) Check your program by making sure that the value approaches about 2.718 as n increases.

You can also graph your results by typing `./interest > interest.dat` and then using *gnuplot* to graph the data. To do this, start *gnuplot* and type `plot "interest.dat" with linespoints`. The result should look something like Figure 4.13.

6. Write a program (call it `baselpi.cpp`) that uses a "`do-while`" loop to sum up the terms of the series:

$$s = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots$$

Notice that the terms keep getting smaller and smaller. Keep adding terms until you come to a term that's less than 10^{-6} (include this

[8] e is perhaps the second most important mathematical constant, after π. If we think of π as the "circle constant", we might think of e as the "growth constant". It appears in equations describing growth and decay in every area of science. For more information, see this Numberphile video by James Grime: https://www.youtube.com/watch?v=AuA2EAgAegE

term in your sum). Print out the sum and the number of terms, clearly identifying which is which. Your program should also use this sum to print an estimate of the value of π. How can it do this? Read on!

Portrait of Leonhard Euler (1707-1783).

Source: Wikimedia Commons

This is a famous problem in the history of mathematics, known as the "Basel Problem[9]". Leonhard Euler was the first to solve this problem, finding that the sum of this series approaches the value $\pi^2/6$. This provides a way to check your program: Multiply the sum by 6 and take the square root. You should get a number that is approximately equal to π.

Hint: When C divides one integer by another, it assumes that you want the answer to be an integer, too. So, if you type `1/i`, where `i` is an integer, C will chop off any decimal places in the answer. If you want to preserve those decimal places, type `1.0/i` instead. This gives C a hint that you want to save things after the decimal place.

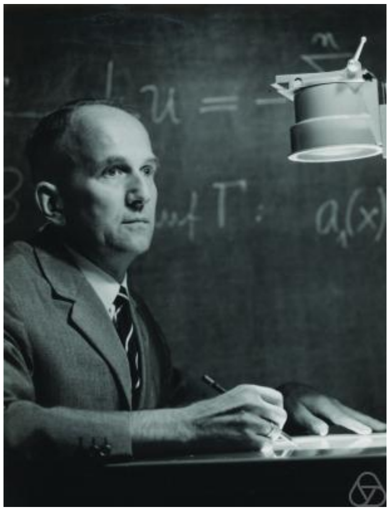
7. Many people think that everything in mathematics is boring, and that there aren't any mathematical discoveries remaining to be made. Nothing could be farther from the truth. Just as there are still plenty of unanswered questions in physics (for example: What is dark matter?) there are also lots of unanswered questions in math. One unsolved mathematical mystery is called the *Collatz conjecture*[10], named after German mathematician Lothar Collatz. Let's write a program that illustrates the property of numbers that Collatz observed.

   Make a program named `lothar.cpp` that asks the user to enter a starting number that's an integer greater than 1. After the number has been entered, the program should have a "`while`" loop that does the following:

   - If the number is *even*, divide it by 2.
   - If the number is *odd*, multiply by 3 and add 1.

   The loop should keep doing this for as long as the result is not equal to 1. Each time around the loop, print the current result. For example, if the user enters the number 5, the program should print:

```
16
8
4
2
1
```



Lothar Collatz (1910-1990)

Source: Wikimedia Commons

   Hint: You can find out whether a number is even by using the

[9] See Wikipedia for much more information.

[10] See https://www.youtube.com/watch?v=5mFpVDpKX70 and https://en.wikipedia.org/wiki/Collatz_conjecture.

modulo operator (`%`). For example, if `i%2` is zero, then `i` is even.

You should find that any number you enter will generate a sequence that ends in 1. Collatz speculated that this was always true for all starting numbers, but nobody has ever been able to prove it. The Collatz conjecture has been tested by computers for all numbers up through 10^{60} and found to be true for each of them, but there might be some huge number out there somewhere that doesn't obey this rule. Nobody knows.

8. Imagine that your algebra teacher has asked you to simplify the expression $12x + 438$. You suspect that there's some common factor of 12 and 438 that you could pull out, but how can you find it? Fortunately, the ancient Greek mathematician Euclid provided us with a simple recipe for finding the greatest common factor of two numbers[11]. Let's call the two numbers n_1 and n_2. Euclid's method works like this:

   1) Divide n_1 by n_2 and find the remainder.

   2) Now make n_1 equal to n_2, and make n_2 equal to the remainder.

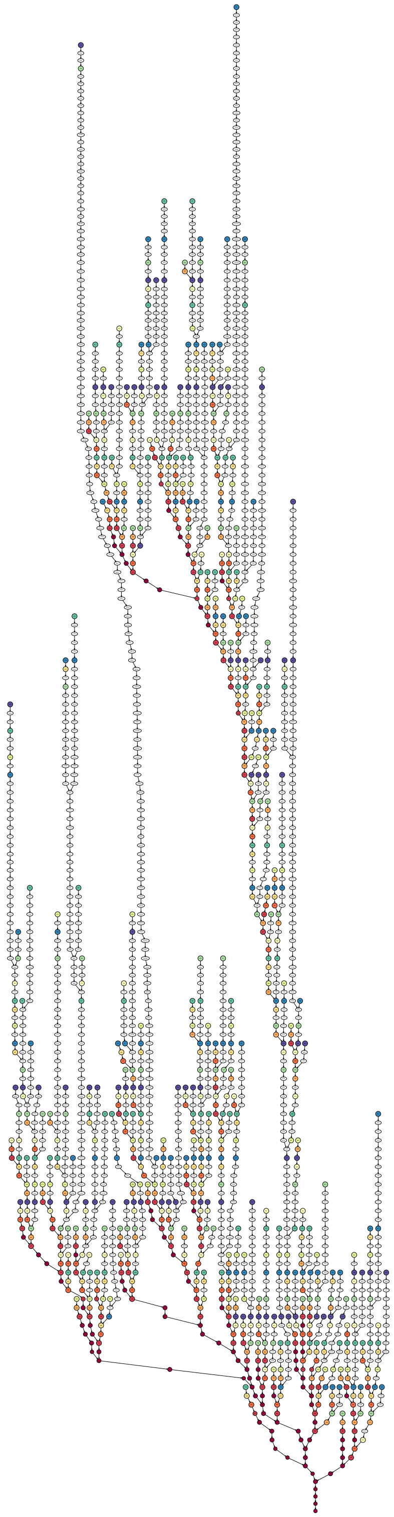
   3) keep repeating steps 1 and 2 until you get to a remainder of zero. At this point, the value of n_1 will be the greatest common factor of the original numbers.

   Write a program named `gcf.cpp` that uses a "`do-while`" loop to find the greatest common factor of two numbers by using Euclid's method. The program should start by asking the user for two integers. When you run the program, it should look something like this:

```
Enter first number: 12
Enter second number: 438
GCF is 6
```

   Hint 1: Remember that the `%` operator gives you the remainder after division.
   Hint 2: If the remainder is `rem`, your loop should continue for as long as `rem != 0`.

9. Write a program named `sumto2.cpp` that uses a `do-while` loop to sum up the terms of the series:



This graph shows the path taken by each of the integers up to 1,000 as they work their way through the Collatz process on their way to 1. As you can see, the paths form a pretty shape, like coral.

Source: Wikimedia Commons

[11] This is also sometimes called the "greatest common divisor" or "greatest common denominator".

$$s = \frac{1}{1} + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \cdots$$

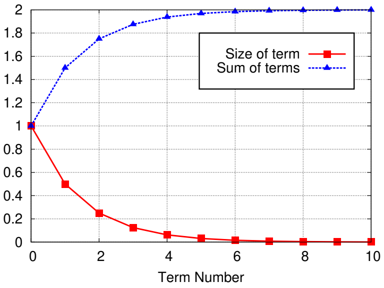


Figure 4.14: In the `sumto2.cpp` program, as we add more terms, each term becomes smaller and their sum converges toward 2.

Notice that the denominators of the terms start with 1, and each denominator is two times as large as the preceding one. Your program should keep adding terms until it comes to one that's smaller than 10^{-9} (include this term in your sum).

If we could add up an infinite number of such terms the sum would be exactly 2. Since each term in the series is substantially smaller than the preceding term, your program should show a sum that's approximately 2.

As we saw in Chapter 3 it's possible to tell C how many decimal places we want to show when printing a number. Inside your program's `do-while` loop, put a statement like this that prints the number of terms so far, the value of each term, and the current sum after adding that term:

```
printf ("%d %.20lf %.20lf\n", nterms, term, sum);
```

The "`.20`" between `%` and `lf` tells the program to print twenty digits after the decimal point. By watching how the terms change, we can see them get smaller and smaller, and we can see the sum get closer and closer to 2.

Hint: To prevent your program from chopping off numbers after the decimal point, use `double` variables to hold the values of the denominators, the terms in the series, and the sum.

10. Imagine you have a bag containing a red marble, a green marble, and a blue marble. You close your eyes, reach in and pull out two marbles. What are the possible outcomes? Well, you could have either:

Boy with marbles.

Source: Wikimedia Commons

- A red marble and a green marble,
- A green marble and a blue marble, or
- A red marble and a blue marble.

So there are three possible results.

We can calculate the number of possibilties for any number of marbles in the bag (let's call that number n), and any number of marbles pulled out (let's call that k). The number of possibilities is given by:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

where $\binom{n}{k}$ is the called the "binomial coefficient" and is read like "n choose k". It tells you how many ways there are to choose k things out of a collection of n things.

As you can see, the calculation involves factorials. C doesn't have a factorial function, but it has something else that can be used in its place. It's called `tgamma`, the "Gamma Function", and it can be used to give the value of factorials[12]. One tricky bit: For historical reasons, $n! =$ `tgamma(n+1)`. So, to get 3! we'd use `tgamma(3+1)`. **Note:** `tgamma` returns a `double` value, so it's important to put the results of calculations using this function into `double` variables.

Now that you know all of that, write a program named `marblechoice.cpp` that asks the user for n and k then prints the value of $\binom{n}{k}$. You can check your program with $n = 7$ and $k = 3$: the program should tell you that there are 35 possible outcomes. (Figure 4.15 shows a few more examples.) Make sure your program checks whether k is bigger than n and gives an appropriate error message in that case. (You can't pull out more marbles than there are in the bag!)

n	**k**	$\binom{\mathbf{n}}{\mathbf{k}}$
2	1	2
2	2	1
3	1	3
3	2	3
3	3	1
7	1	7
7	2	21
7	3	35
7	4	35
7	5	21
7	6	7

Figure 4.15: A few values of $\binom{n}{k}$.

11. The cosine function can be expressed as an infinite sum of terms[13]:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots$$

Each term in the series (including the first one) is of the form:

$$\frac{(-1)^n}{(2n)!}x^{2n}$$

Where $n = 0$ for the first term, $n = 1$ for the second, and so forth.

Write a program named `mycos.cpp` that asks the user for an angle, in radians, and calculates an approximate value for the cosine of that angle by adding up terms of this series until it comes to a term whose absolute value[14] is less than 10^{-9}.

The program should use a "`do-while`" loop and an integer variable named `n` which starts with a value of zero.. Each time around the loop, the program should calculate the value of the current term in the series and add it to the current sum of all the terms[15]. (Let's store the value of the current term in a variable named `term` and the value of the sum in a variable called `sum`.)

[12] Be careful! Some versions of C also have a function named `gamma`, but that's something different, and should be avoided.

[13] this is called a Taylor series.

[14] We need to look at the absolute value because some of the terms are negative.

[15] Remember to set the sum equal to zero before you start adding to it.

As you can see, the calculation involves factorials. C doesn't have a factorial function, but it has something else that can be used in its place. It's called `tgamma`, the "Gamma Function", and it can be used to give the value of factorials[16]. One tricky bit: For historical reasons, $n! =$ `tgamma(n+1)`. So, to get 3! we'd use `tgamma(3+1)`.

Each time you calculate the value of a new term, print the current values of `n`, `term`, and `sum` like this:

```
printf( "%3d %23.20lf %23.20lf\n", n, term, sum );
```

The "`23.20`" between `%` and `lf` tells the program to print numbers using 23 characters, with twenty digits after the decimal point. This makes them line up nicely in columns. By watching how the terms change, we can see them get smaller and smaller, and we can see the sum get closer to the true value of $\cos x$.

Add 1 to the value of `n` each time you go around the loop. At the bottom of the loop (the "`while`" statement) the program should check to see if the absolute value of the current term is still greater than 10^{-9}. The C math function `fabs` gives you the absolute value of a number.

Serpentine walls at the University of Virginia approximate the shape of sine (or cosine) curves.

Source: Wikimedia Commons

If you give your program a value of 3.14 radians, its output should look something like this:

```
  0  1.00000000000000000000  1.00000000000000000000
  1 -4.92980000000000018190 -3.92980000000000018190
  2  4.05048800666666775072  0.12068800666666756882
  3 -1.33120638501768939754 -1.21051837835102182872
  4  0.23437790131643590485 -0.97614047703458595162
  5 -0.02567635950910591297 -1.00181683654369191316
  6  0.00191786844103015606 -0.99989896810266176708
  7 -0.00010389788835813697 -1.00000286599101984031
  8  0.00000426829841689953 -0.99999859769260290854
  9 -0.00000013752848062504 -0.99999873522108351231
 10  0.00000000356835738834 -0.99999873165272612496
 11 -0.00000000007615276300 -0.99999873172887887574
```

So what good is all this? This is actually the way your calculator finds the value of a cosine when you press the "cos" button. It doesn't have any magical mathematical knowledge of trigonometry. It just uses a Taylor series to find an approximate value.

[16] Be careful! Some versions of C also have a function named `gamma`, but that's something different, and should be avoided.

12. The sine function can be expressed as an infinite sum of terms:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots$$

Each term in the series is of the form:

$$\frac{(-1)^n}{(2n+1)!}x^{2n+1}$$

Using these facts, and the description given in Problem 11, write a program named `mysin.cpp` that gives an approximate value for the sine of an angle. Other than the formula for the terms in the series, this program should be exactly like the program described in Problem 11.

13. The exponential function (e^x) can be expressed as an infinite sum of terms:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

Each term in the series is of the form:

$$\frac{x^n}{n!}$$

where n starts with a value of zero and $0!$ is, by definition, equal to one.

Using these facts, and the description given in Problem 11, write a program named `myexp.cpp` that gives an approximate value for e^x. Other than the formula for the terms in the series, and the fact that it will ask for a value for x instead of an angle, this program should be exactly like the program described in Problem 11.

14. Gottfried Leibniz is one of several people who independently discovered that

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

This relationship gives us a way to find an approximate value for π. The right-hand side of the equation above is an infinite series of terms of the form:

$$\frac{(-1)^k}{2k+1}$$

Since the denominator gets bigger as k increases, each term in the series is a little smaller than the preceding one. To get an approximate value for this infinite sum, we can just keep adding terms until the terms get so small that we feel like we can ignore the rest of them.

In 2022, Emma Haruka Iwao took the world record for calculating digits of π, calculating its value to 100 trillion digits.

Source: Wikimedia Commons

Then we can get an approximate value for π by multiplying this sum by 4 (see the left-hand side of the equation at the beginning).

Write a program named `leibnizpi.cpp` that uses this method to calculate an approximate value for π. The program should use a "`do-while`" loop and an integer variable named `k` which starts with a value of zero.. Each time around the loop, the program should calculate the value of the current term in the series and add it to the current sum of all the terms[17]. (Let's store the value of the current term in a variable named `term` and the value of the sum in a variable called `sum`.)

Each time you add a term to the series, print the current values of `k`, `term`, and `4.0*sum` like this:

```
printf("%d %.20lf %.20lf\n", k, term, 4.0*sum);
```

The "`.20`" between `%` and `lf` tells the program to print twenty digits after the decimal point. By watching how the terms change, we can see them get smaller and smaller, and we can see `4.0*sum` get closer and closer to π.

Add 1 to the value of `k` each time you go around the loop. At the bottom of the loop (the "`while`" statement) the program should check to see if the absolute value of the current term is still greater than 10^{-5}. The C math function `fabs` gives you the absolute value of a number. (We need to look at the absolute value of the term because some terms will be negative.)

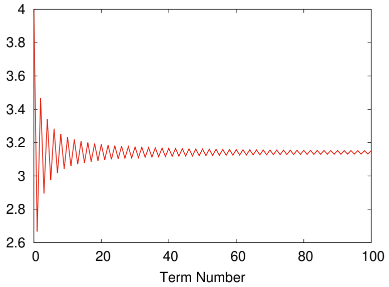


Figure 4.16: A graph of the output of the `leibnizpi.cpp` program, showing how the approximation zooms up and down, eventually settling near the value of π.

When you run your program, notice that the values converge toward π *very* slowly. It should take about 5,000 terms before you get to one that's less than 10^{-5}. That shows that this isn't a very efficient way to determine the value of π. Also notice that the value of `4.0*sum` bounces up and down: sometimes it's larger than π and sometimes it's smaller. We can see this by graphing the program's output. Run your program again, like this:

```
./leibnizpi > leibnizpi.dat
```

and then start up *gnuplot* and give it these commands:

```
set xrange [0:100]
plot "leibnizpi.dat" using 3 with lines
```

The "`using 3`" tells *gnuplot* to plot the numbers from column 3 of

[17] Remember to set the sum equal to zero before you start adding to it.

your program's output. The "`set xrange [0:100]`" tells *gnuplot* to zoom in on the first part of the program's output, when the terms are changing a lot. The result should look like Figure 4.16.

15. Musicians usually organize sound into octaves, where an octave is a range that starts at some base frequency and goes up to twice that starting frequency. In music from the European tradition, an octave is typically divided into twelve sections, each starting with a different frequency, or "tone".

    Figure 4.17 shows a set of twelve tones that starts with a base frequency of 440 Hz[18]. Each of the tones is obtained by multiplying the preceding term by a constant ratio, which we'll call `r`. If we want to get to twice the original frequency after twelve steps, that means that `r` has to be equal to $\sqrt[12]{2}$ (the twelfth root of two)[19].

    Write a program named `12tone.cpp` that prints the data shown in Figure 4.17. To do this, the program should devine a variable named `r`, and we'll need to set it to $\sqrt[12]{2}$. (**Hint:** Use the `pow` function to raise 2 to the `1.0/12` power.) Then the program should have a "`for`" loop that prints tone number (0 to 12), frequency, and the ratio of frequency to base frequency for each of the tones. The results should look like the numbers in Figure 4.17.

Tone	Frequency	Frequency/Base
0	440.00	1.00
1	466.16	1.06
2	493.88	1.12
3	523.25	1.19
4	554.37	1.26
5	587.33	1.33
6	622.25	1.41
7	659.26	1.50
8	698.46	1.59
9	739.99	1.68
10	783.99	1.78
11	830.61	1.89
12	880.00	2.00

Figure 4.17: Twelve tones, starting with a base frequency of 440 Hz. The table also shows a thirteenth tone, which is just twice the base frequency and is the beginning of the next octave.

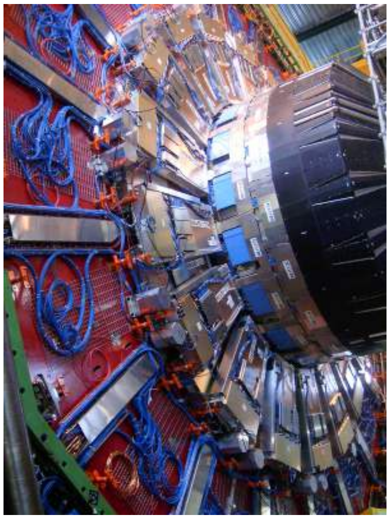
[18] This is the frequency of the A above middle C on a piano.

[19] This is called the 12-tone even-tempered system of tuning.

5. *Reading and Writing Files*

5.1. Introduction

CERN's Large Hadron Collider produces mountains of data: about a gigabyte (10^9 bytes) per second. That's enough to fill a couple of hundred laptop-sized disks per day! This data is saved in files, and these files are distributed around the world for analysis.



A part of the CMS detector, at CERN's Large Hadron Collider.

Source: Wikimedia Commons

Early computers read data from punched cards, or from paper tape with holes punched into it. The pattern of holes on each card was a code that represented numbers or letters. "Keypunch operator" was a job much-advertised in the help-wanted section of the newspaper. A keypunch machine was similar to a typewriter. As the operator typed, holes were punched in the appropriate places on the card. Some keypunch machines also typed the words onto the card, so you could look at it and easily see what was encoded on it (although many programmers became quite adept at reading the holes themselves).

A keypunch machine in the basement of the UNC Physics building. As late as the 1980s, undergraduates would flock there nightly to punch cards for programming projects.

Source: UNC-Chapel Hill Computing History photo collection

Each punched card could store about eighty bytes of information. If digital cameras had existed at that time, storing a single photo would have required tens of thousands of cards. As computers became faster and capable of dealing with larger data sets, new storage technologies had to be developed. One of these was magnetic media, first in the form of tapes and later disks. Early reel-to-reel tapes of the type introduced by IBM in the 1960s could hold several tens of megabytes: enough for a few photographs from a modern camera. Removable "diskettes" (also called "floppy disks") were developed in the 1970s and 80s. These couldn't hold as much data as tapes, but they were were convenient for storing a few spreadsheets or word-processing documents. "Hard disks", of the type still in use today, can hold several terabytes (10^{12} bytes) of data. That's enough to hold hundreds of thousands of digital photos.

A magnetic tape library at the National Oceanographic Data Center.

Source: Wikimedia Commons

As we've learned in earlier chapters, computers store data in the form of ones and zeros. A "file" on a disk is just a collection of ones and zeros, with a name attached to it so we can find it when we need it. In this chapter, we'll learn how to write data to files and read data from files.

A very famous broken file cabinet. This is the cabinet that was broken into in the Watergate Hotel, at the behest of the Nixon administration. It now resides in the Smithsonian's National Museum of American History.

Source: Wikimedia Commons

5.2. Writing Files

Until now, we've used the `printf` function to send output to the computer's screen. If we want to write things into a file instead, we can use another function named `fprintf` (for "file `printf`"). Before we can do that, though, we have to do a little preliminary work.

Writing to a file isn't quite as simple as writing to the screen. For one thing, we can usually assume that there's a screen to send our output to, but the file might not exist. If it doesn't exist, do we want to create it, or just give the user an error message? If the file exists already, do we want to replace its contents with something new, or do we want to add content after the end of whatever's already there?

We can control all of these options with the `fopen` function. The `fopen` function "opens" a file and makes it ready for reading or writing.

A companion to `fopen` is the `fclose` function. This makes sure that all data has completely been written to a file. Although programs will usually do this for you automatically when they finish running, it's good practice to explicitly use the `fclose` function to "close" a file when you're done with it.

The `fopen` function returns a value that can be used to identify the file you've opened. This identifier is called a "file handle", since it's like a handle by which you can grab the file when you need it.[1] As you'll see, there's a new kind of variable that we use just for storing file handles.

When you use the `fprintf` function to print something into a file, you tell `fprintf` which file to use by giving it a file handle.

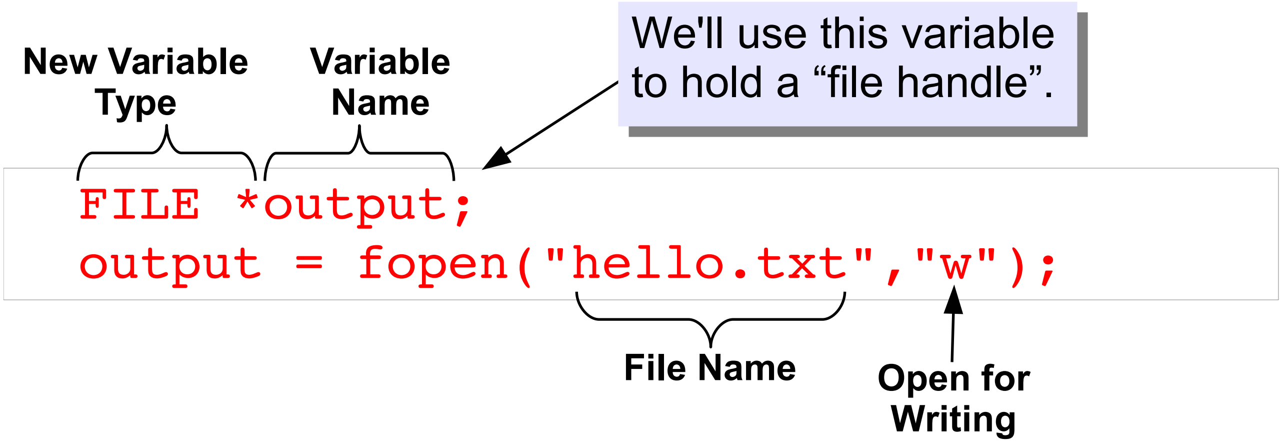
Program 5.1 is a very simple example showing how to open a file, write something into it, and then close it. The program writes the words "Hello File!" into a file named `hello.txt`.

[1] This identifier is sometimes referred to as a "file descriptor" or "file pointer". These are all the same thing.

Program 5.1: hellofile.cpp

```
#include <stdio.h>
int main () {
  FILE *output;
  output = fopen("hello.txt","w");

  fprintf( output, "Hello File!\n");

  fclose( output );
}
```

Even though Program 5.1 is short, there's a lot going on in it. Let's look at some of the parts individually. First, let's look a the fopen statement:



```
FILE *output;
output = fopen("hello.txt","w");
```

Figure 5.1: Structure of an fopen statement.

As you can see from Figure 5.1, fopen takes two arguments: the name of the file to be opened, and a second argument that specifies how we're going to use the file. For example, we can say that we want to read ("r"), write ("w") or append ("a") to the file. There are also other options. See Figure 5.3 for some of them. Usually, you'll only need "r" or "w".

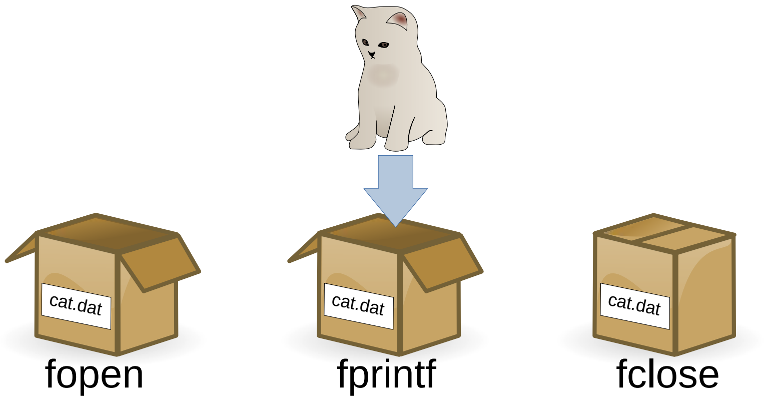
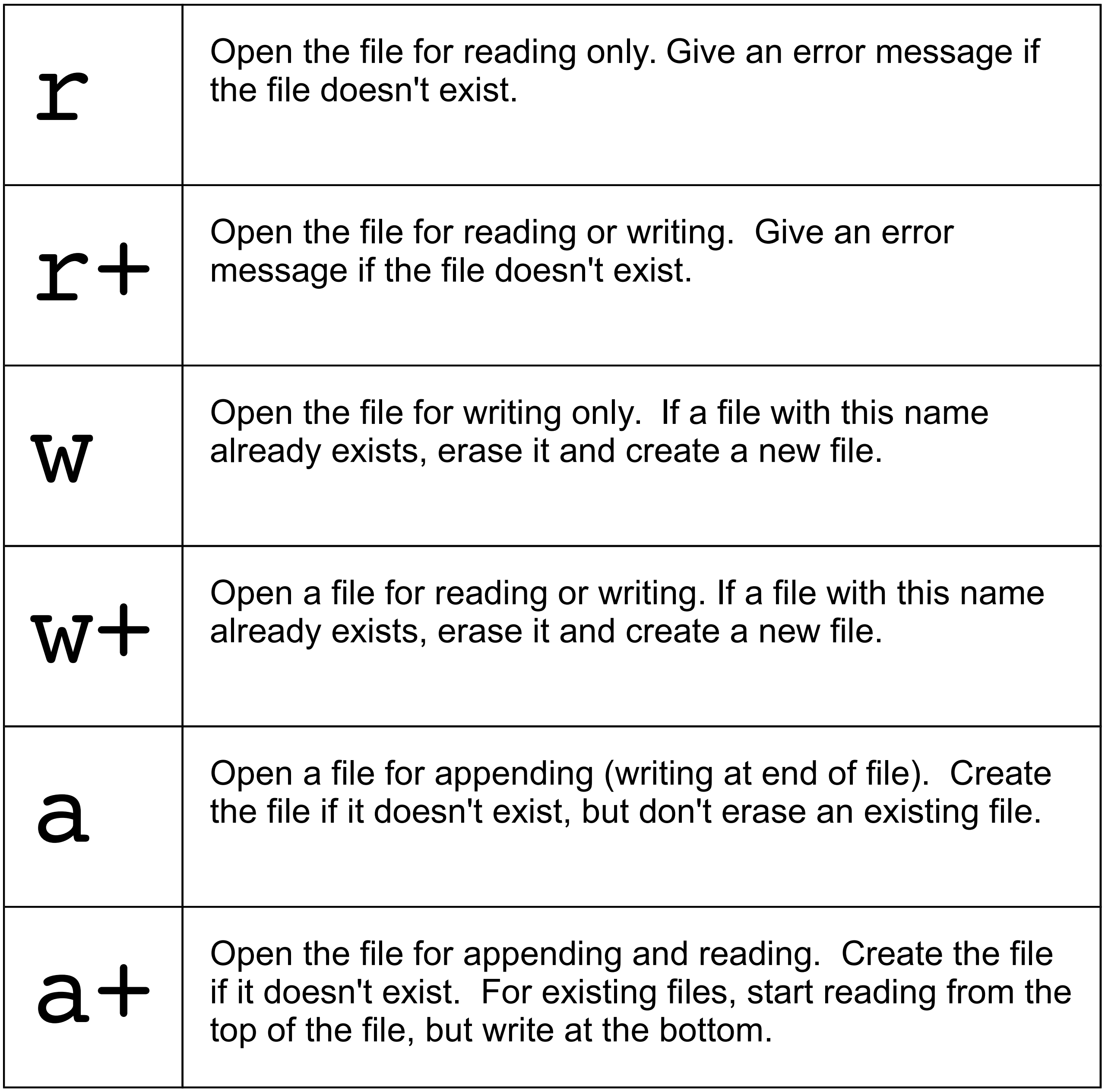


Figure 5.2: If you think of a “file” as a box that can contain some data, then fopen opens the box, fprintf stuffs some data into the box, and fclose closes the box.



`r`	Open the file for reading only. Give an error message if the file doesn't exist.
`r+`	Open the file for reading or writing. Give an error message if the file doesn't exist.
`w`	Open the file for writing only. If a file with this name already exists, erase it and create a new file.
`w+`	Open a file for reading or writing. If a file with this name already exists, erase it and create a new file.
`a`	Open a file for appending (writing at end of file). Create the file if it doesn't exist, but don't erase an existing file.
`a+`	Open the file for appending and reading. Create the file if it doesn't exist. For existing files, start reading from the top of the file, but write at the bottom.

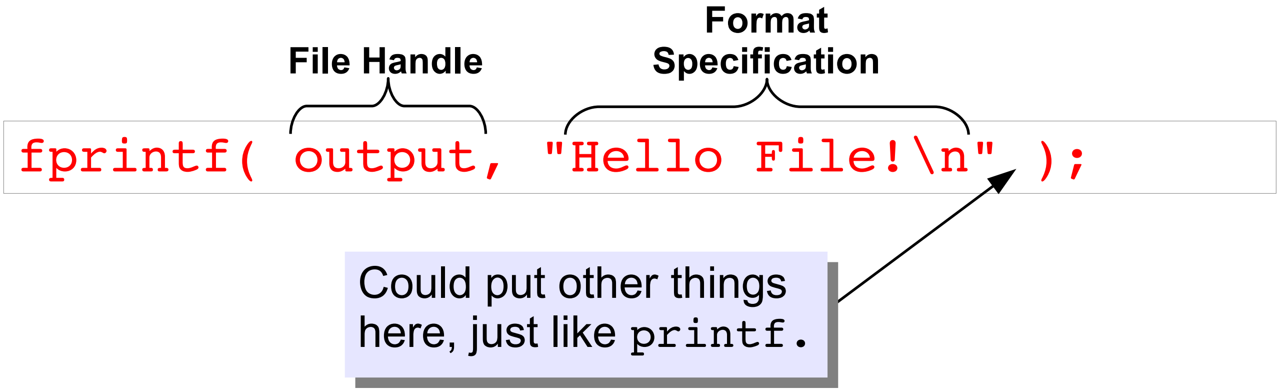
Figure 5.3: Various ways that fopen can open a file.

The Writing Master, by Thomas Eakins.

Source: Wikimedia Commons

The `fopen` function returns a file handle, which we can capture in a variable for later use. In Program 5.1 we name this variable "`output`", but it can have any name you want to give it. This is a new kind of variable, unlike the `int` and `double` variables we've been using to store numbers. It's a special type of variable just for storing file handles. Just as we might define an integer variable by saying "`int i`", we define this new variable by saying "`FILE *output`". Note the asterisk here is part of the file type. The type of this variable isn't `int` or `double`, it's "`FILE *`".

Once we've stored the file handle in a variable, we can use it to read from a file or write to a file. The `fprintf` function is like `printf`, except that it takes one extra argument: a file handle. In Program 5.1 we use the `fprintf` function to write the text "Hello File!" into the file `hello.txt`, which we've previously opened with `fopen`. We've specified this file by giving `fprintf` the file handle "`output`". If we wanted to, we could open several different files and write different things into each of them. In that case, we'd pick which file we wanted to use by giving the appropriate file handle to the `fprintf` function.



```
fprintf( output, "Hello File!\n" );
```

Figure 5.4: Structure of an `fprintf` statement.

Finally, Program 5.1 uses the `fclose` function to make sure everything has been written to the file before the program finishes.

Exercise 26: Hello File!

Create, compile and run Program 5.1. When you run the program, you shouldn't see any output since it's being sent into a file instead of to the screen. How can you tell if the program did the right thing?

First of all, look to see if there's a new file. The `ls` command will show you a list of your files. Do you see a file called `hello.txt`?

Next, take a look inside the file by typing `nano hello.txt`. Does it contain the text "Hello File!" as it should?

The word "hello" wasn't commonly used until the invention of the telephone. There was initially some disagreement about the proper form of greeting on the new device. Alexander Graham Bell favored "Ahoy!", and some people advocated the jauntier variant "Hoy, Hoy!". Eventually, we settled on "Hello!", and it was so much identified with the device that early telephone operators were referred to as "Hello Girls".

Source: Wikimedia Commons

But what about...?

In earlier chapters, we've seen that we can redirect the output of our programs into a file by appending > followed by a file name when we run the program (as we did when plotting the output of our `gutter` program in Chapter 2). You can alternatively use » to append some output at the end of an existing file. For example, you could do the following:

```
./gutter > gutter.dat
./gutter >> gutter.dat
./gutter >> gutter.dat
```

The first command would create a new file called `gutter.dat` and write the program's output into it. The next command would run the program again, and append the output onto the end of the existing file. The last command appends even more output onto the file.

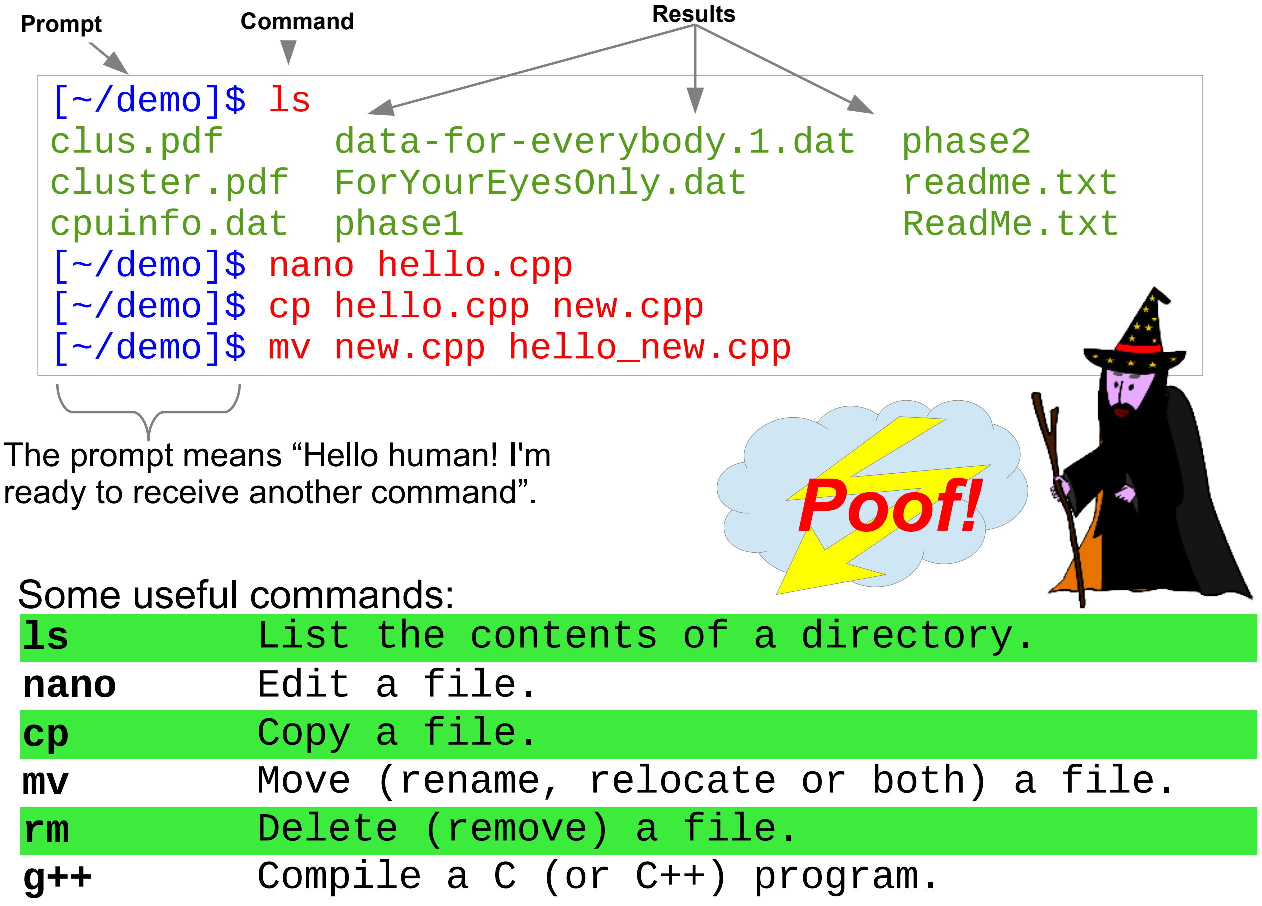
If we can use > or » to redirect a program's output into a file, why would we want to make our C programs write files in any other way? There are at least a couple of reasons:

- Sometimes we want to send some output to the screen and some to a file. Think about a program that asks the user for some input, and then writes out some data. Text that says "Please enter your age" should go to the screen, but we might want the rest of what the program writes to go into a file.
- Sometimes a program needs to write more than one file. Think about a program that sorts data into several categories, and writes each category to a different file. Imagine the program that Santa uses to sort kids into `naughty.dat` and `nice.dat`.

Be careful when using > to send a program's output into a file. If you type the wrong file name, you could accidentally write over a file you want to keep!

5.3. Some Useful Commands for Managing Files

In the exercise above we saw the ls command, and we've been using the commands nano, g++, and gnuplot for a while now. Figure 5.5 summarizes some commands that you might find useful when working with files.



Prompt
Command
Results
The prompt means “Hello human! I'm ready to receive another command”.
Poof!

```
[~/demo]$ ls
clus.pdf      data-for-everybody.1.dat  phase2
cluster.pdf   ForYourEyesOnly.dat       readme.txt
cpuinfo.dat   phase1                    ReadMe.txt
[~/demo]$ nano hello.cpp
[~/demo]$ cp hello.cpp new.cpp
[~/demo]$ mv new.cpp hello_new.cpp
```

Some useful commands:

ls	List the contents of a directory.
nano	Edit a file.
cp	Copy a file.
mv	Move (rename, relocate or both) a file.
rm	Delete (remove) a file.
g++	Compile a C (or C++) program.

Figure 5.5: Some useful commands for managing files.

Source: Openclipart.org

As we saw in the exercise above, you can use the ls command to show us a list of our files.[2] The cp ("copy") command can be very useful in cases where you want to write a new program that's similar to one you've written in the past. You can make a copy of the old program, with a new name, and then modify the copy as needed.

When entering commands at the command line, notice that the computer will usually put a "prompt" at the beginning of each new line. This is some text that might tell you what folder you're working in, or what the computer's name is. The text will vary depending on the type of computer and its configuration. In any case, think of the prompt as the computer's way of saying "OK, I'm ready for you to give me a new

[2] "ls" is just an abbreviation for "list". As we've seen before, programmers are sometimes lazy typists.

command now."

Although the commands in Figure 5.5 have strange names, you might think of them as wizardly incantations like Harry Potter's "lumos!". By invoking these arcane spells you can cause the computer to do useful things for you.

5.4. Infinite Loops

Sometimes a program doesn't know how much data you're going to give it. Consider Program 5.2 for example.

Program 5.2: input.cpp

```
#include <stdio.h>
int main () {

  int nsiblings;
  int nperson = 0;

  FILE *output;
  output = fopen("siblings.txt","w");

  printf ("Enter the number of siblings, or -1 to quit.\n");

  while ( 1 ) {
    printf ( "Number of siblings for person %d: ", nperson );
    scanf ( "%d", &nsiblings );
    if ( nsiblings < 0 ) {
      break;
    }
    fprintf( output, "%d %d\n", nperson, nsiblings );
    nperson++;
  };

  printf ("Thank you!\n");

  fclose( output );

}
```

Imagine you're collecting data about how many siblings your classmates have. Program 5.2 prompts you to enter the number of siblings each individual has, and saves the data into a file called `siblings.txt`.

The address of Apple's corporate headquarters is "1 Infinite Loop".

Source: Wikimedia Commons

Notice how the program uses the "`while`" loop. As we saw in Chapter 4, a "`while`" loop keeps going for as long as the condition in parentheses is true. Here, the value in parenthesis is just "`1`". Is that true or false?

When a C program comes to a condition statement after an "`if`" or "`while`", the computer converts the condition into a number. If the condition statement is false, the number is zero. Any other number means the statement is true. The "`if`" or "`while`" then uses this number to decide what to do. If we use the number `1` as the condition, it will always be true, so the "`while`" statement in Program 5.2 will keep looping forever unless we somehow tell it to stop. This is called an "infinite loop".

Program 5.2 uses an infinite loop because it doesn't know beforehand how many people you're going to survey. It just keeps asking for more data until you explicitly tell it you're done. When you've collected all of your data, you signify this by giving `-1` as the number of siblings. This causes the `break` statement to be acted upon, and the loop terminates.

Our program assumes that nobody really has a negative number of siblings. How could that even happen? Antimatter??

Source: Wikimedia Commons

Infinite loops like this are often used when a program needs to keep doing something until the user tells it to stop. For example, there's an infinite loop underneath the operating system on your computer. It waits for mouse clicks, keystrokes, and other interesting events, and examines them to find out what you're asking it to do. At some point, you may tell the computer to shut down, causing the operating system to clean things up and break the loop.

Exercise 27: Collecting Data

Create, compile and run Program 5.2. Enter some data from your friends and neighbors, or just make something up. Enter at least ten numbers. When you're done, enter "-1" to stop the program.

Now use *nano* to look at the program's output file: Type "`nano siblings.txt`". Does it look like what you expected?

Exit from *nano*, then start up *gnuplot*. Plot the data you've collected by giving gnuplot the command:

```
plot "siblings.txt" with boxes
```

The result should look something like Figure 5.6. The phrase "`with boxes`" tells gnuplot to draw boxes instead of just plotting points.

Depending on how many points you entered, you may find that *gnuplot* chops off part of the first box. You can fix this by explicitly telling *gnuplot* where you want the x axis to start. To do this, type:

```
set xrange [-1:]
```

and then type "`replot`".

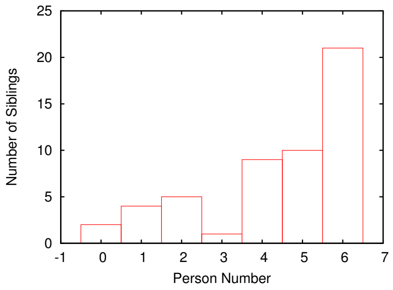


Figure 5.6: Example sibling data, plotted with *gnuplot*.

5.5. Producing Data Files

Sometimes programs write data, and sometimes they read data. It's often the case that data written by one program will be read by a different program. Think about the experiments at CERN. During an experiment, programs collect the data from particle detectors and write the data into files. Later, perhaps at a university elsewhere, someone uses a different program to read the data files and analyze them.

Let's create a pair of programs that produce and consume data. The first one will write some data into a file, and the second will read the data and do something useful with it. The data will involve a simple physics problem, but don't worry if you don't understand the physics.

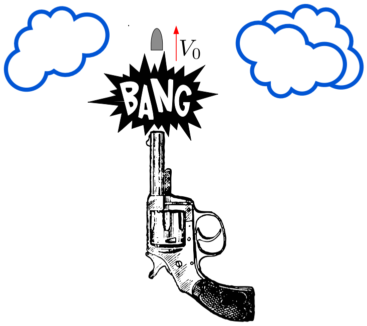


Figure 5.7: The scenario behind Program 5.3

Imagine that you fire a gun straight up into the air. The bullet leaves the gun's muzzle at approximately 700 meters per second. As it rises, gravity slows it until eventually it stops rising and begins to fall. Assuming a constant deceleration due to gravity, the velocity of the bullet at any time after it's fired would be:

$$V = V_0 - gt$$

where t is the elapsed time in seconds, V_0 is the bullet's initial velocity, in meters per second, and g is the acceleration due to gravity near the earth's surface, which is about 9.8 m/s^2. Because of the minus sign, the bullet's velocity gets smaller and smaller as time passes, until it eventually reaches zero, and then it becomes negative (meaning that the bullet has started falling back to earth).

The height of the bullet at any time will be:

$$h = V_0 t - \frac{1}{2} g t^2$$

if we assume that the bullet starts from a height of zero.

We assume that the acceleration of gravity is a constant, which is approximately true if we don't get too far from the surface of the earth. In Georges Melies' 1902 film *Le Voyage dans la Lune* six men are fired to the moon inside a large artillery shell. Needless to say, our approximation would not hold true in this situation.

Source: Wikimedia Commons

Program 5.3 calculates the bullet's velocity and height once per second during the first one hundred seconds of its flight, and writes those values into a file for later analysis.

Program 5.3: bullet.cpp

```
#include <stdio.h>
#include <math.h>
int main () {

  int i;
  double t = 0.0;
  double v;
  double h;
  double v0; // meters per second.
  double delta_t = 1.0; // seconds.
  double g = 9.8; // meters/second.
  FILE *output;

  printf ( "Enter initial velocity (m/s): " );
  scanf ( "%lf", &v0 );

  output = fopen("bullet.txt","w");

  for ( i=0; i<100; i++ ) {
    v = v0 - g*t;
    h = v0*t - 0.5*g*pow(t,2);
    fprintf( output, "%lf %lf %lf\n", t, v, h );
    t += delta_t;
  };

  fclose( output );

}
```

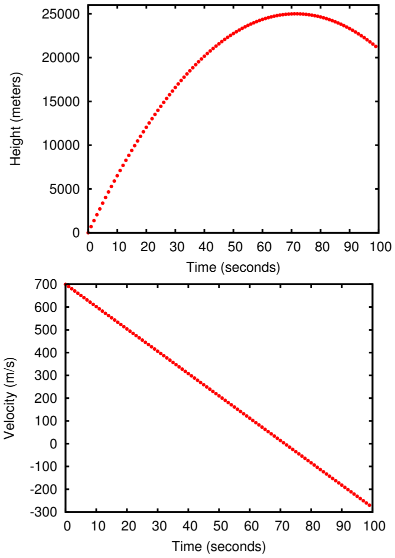


Figure 5.8: A bullet's height and velocity as a function of time, for a starting velocity of 700 m/s.

Notice that we've added some comments beside the definitions of our variables to remind us what units we're using. Comments like this can be very helpful if someone else needs to understand your program.

Exercise 28: Fire At Will!

Create, compile and run Program 5.3. It should ask you for an initial velocity. Use 700 m/s. After the program finishes, use the "`ls`" command to check that the output file, `bullet.txt`, has been created. Take a look inside the file with *nano* by typing "`nano bullet.txt`". There should be three columns of data, for time, velocity, and height.

Now exit from *nano* and use *gnuplot* to plot the bullet's height versus elapsed time, by giving *gnuplot* this command:

```
plot "bullet.txt" using 1:3
```

You should see a graph that looks like top graph in Figure 5.8. Try to identify the bullet's maximum height, and the time at which it reaches this height.

If you have time, you can also graph the bullet's velocity as a function of time by giving gnuplot this command:

```
plot "bullet.txt" using 1:2
```

But what about...?

Notice that Program 5.3 only tracks the bullet for one hundred seconds. The bullet may not reach the ground during that time. What if we wanted the program to track the bullet for as long as it's in the air, and stop when it hits the ground? We could modify the program by replacing the "`for`" loop with a "`do-while`" loop, like this:

```
do {
    v = v0 - g*t;
    h = v0*t - 0.5*g*pow(t,2);
    fprintf( output, "%lf %lf %lf\n", t, v, h );
    t += delta_t;
} while ( h >= 0.0 );
```

5.6. Analyzing a Data File

In the exercise above, you might have found that it was hard to tell exactly where the bullet reached its maximum height by looking at the graph of our data. Analyzing data by hand is tedious and imprecise.

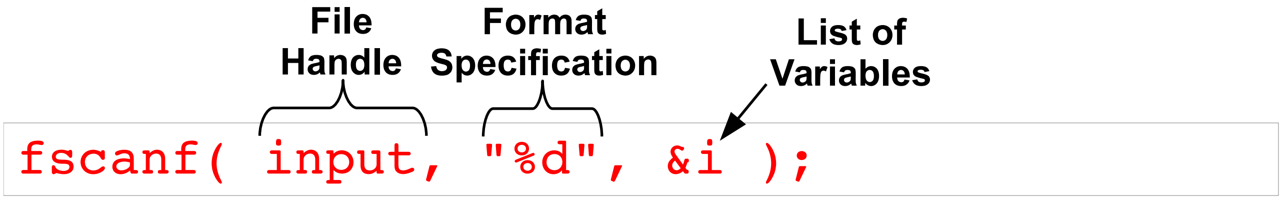
Imagine how much harder it is to analyze the data from a huge experiment like the ones at CERN, where billions of data points are recorded per second!

Even for small experiments, it's often necessary to write computer programs to help us analyze data. Let's write a program that can read the bullet program's output file and find the maximum height for us.

Take a look at Program 5.4 on Page 159. This program does several new things. First of all, it opens the file for reading, instead of writing, by giving an "r" to the fopen function.

Next, notice that Program 5.4 uses an infinite loop (see the "while (1)") to read data from the file. This allows the program to read a file of any length. If we modified our bullet program so that it produced more or fewer lines of data, Program 5.4 would still be able to read the output file.[3]

Each time Program 5.4 goes around its loop, it reads a line from the bullet.txt data file. To do the reading, we use a new function: fscanf. The fscanf function is like scanf, except that it reads data from a file instead of from the keyboard. The first argument we give fscanf is a file handle. This tells fscanf which file we want to read from. In principle, we could open up several different files and choose which one we want to read by giving the appropriate file handle to fscanf. Figure 5.9 shows the structure of a typical fscanf statement.



```
fscanf( input, "%d", &i );
```

Figure 5.9: Structure of an fscanf statement.

Just like scanf, you should always put an ampersand (&) in front of the variable names whenever you read numbers with fscanf, and you should avoid "\n" in the format specification you give fscanf.[4]

Since the program uses an infinite loop, we have to do some sort of test inside the loop to see if we're done yet. In this case, we check to see if fscanf successfully read the number of things we asked it to read. Each time we call fscanf it returns an integer value that indicates how many things it read. If things go well, this number should be equal to the number of variables we ask fscanf to read. For example,

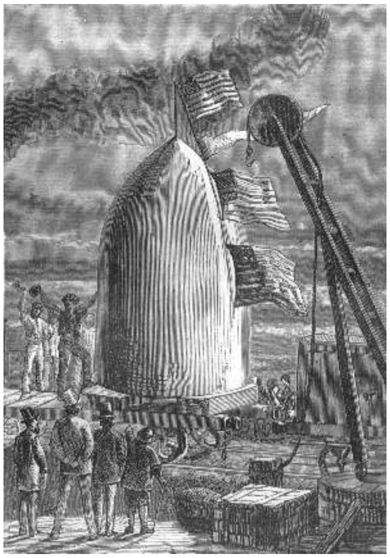
[3] This would be very important if we changed the loop in our bullet program to a "do-while" loop, as in shown in the box above. In that case, we'd never know how many lines of data the program would generate.

[4] See Chapter 3.

if we have a fscanf statement like the one shown in Figure 5.9, the value returned should be 1, since there's only one variable in the list. If we get a different number, that means fscanf couldn't do what we asked it to do. When we're reading data from a file, we'll use this as an indication that we've read all of the data, and it's time to stop reading.

Now that we understand the mechanics of reading a file, how do we find the maximum height in our bullet data? First, we create variable called hmax, in which we'll store the maximum height. After opening our data file, we read it, one line at a time. Each line of the file contains three numbers: the elapsed time since the bullet was shot, the current velocity, and the current height. We initially set hmax equal to the first height value in the file, then each time we read another line from the data file, we look to see if its height is greater than hmax. If it is, we make this height the new value of hmax. When we're done looking at all of the data, hmax should contain the maximum height value.

The program also finds the time at which the maximum height is reached. Whenever the program sets a new hmax value, it also sets the variable tmax equal to the time value that appears on the same line of the data file. When the program finishes, tmax should contain the time at which the maximum height was reached.



Another group of intrepid adventurers who journeyed to the Moon inside an artillery shell. These are from Jules Verne's *From the Earth to the Moon*, as illustrated by Henri de Montaut.

Source: Wikimedia Commons

Exercise 29: Finding the Maximum

Create, compile and run Program 5.4. Does it give you results that match your expectations?

Now try running your earlier bullet program again, this time giving it a different initial velocity, say 600 m/s instead of the 700 m/s you used earlier. Run your readbullet program again to find the new maximum height.

If you pick an initial velocity much higher than 700 m/s, you'll find that your readbullet program will always tell you that the time at maximum height is 100 seconds. This is because our bullet program only tracks the bullet for 100 seconds, and if its initial velocity is too large the bullet will still be rising at the end of this time.

If you have time, look at the new bullet.txt file with gnuplot, as you did before, to see if the maximum height looks like it matches the output of readbullet.

Program 5.4: readbullet.cpp

```
#include <stdio.h>
int main () {
  double t;
  double v;
  double h;

  double hmax;
  double tmax;
  int initialized = 0;

  FILE *input;

  input = fopen("bullet.txt","r");

   while ( fscanf( input, "%lf %lf %lf", &t, &v, &h ) == 3 ) {
    if ( initialized == 0 || h > hmax ) {
      hmax = h;
      tmax = t;
    }
    initialized = 1;
   }

  printf ( "Maximum altitude of %lf after %lf seconds\n", hmax, tmax );

  fclose( input );
}
```

Have we initialized hmax?

Open the file for reading, using "r"

Did we get 3 numbers?

Read lines from the file

Have we found a greater height? (Or do we need to initialize hmax?)

hmax has now been initialized.

5.7. The Perils of Excessive open/close

We saw in Chapter 4 that modern computers are very fast. Adding up the square roots of a billion numbers takes only seconds. But some things take longer than others. In particular, it takes a computer a relatively long time to open or close a file.

THE ABDUCTION.
CP
THE ECLECTIC FILM COMPANY'S
GREAT $ 25.000 PRIZE PHOTO PLAY
THE PERILS OF PAULINE
6TH EPISODE IN 2 PARTS
COPYRIGHTED 1914 ECLECTIC FILM CO.

Eeek!

Source: Wikimedia Commons

We can test this with a program like Program 5.5. Here we have a loop that opens and closes a file a million times. Each time around the loop, the program opens the file, writes some text into it, and closes the file. Before the loop starts, the program saves the current time in the variable tstart. After the loop finishes, we calculate how much time has passed since tstart. The program prints the total time, in seconds, and also prints the time per open/close.

If your computer has an old-fashioned spinning disk this program might take a few minutes to run, with each open/close taking about a millisecond. On a modern solid-state disk each open/close might only take a tenth of a millisecond, but the program will still take several seconds to run. If we increased ntimes to a billion, the program would take a thousand times longer (several hours at least). Compare that with the few seconds it took our earlier test program (Program 4.1) to add up the square roots of a billion numbers. You can see that opening and closing files is much slower than just doing math.

Program 5.5: openclose.cpp

```
#include <stdio.h>
#include <time.h>
int main () {
  int i;
  int ntimes = 100000;
  int tstart;
  double delay;
  FILE * output;

  tstart = time(NULL);

  for ( i=0; i<ntimes; i++ ) {
    output = fopen( "openclose.dat", "w" );
    fprintf( output, "Testing...\n" );
    fclose( output );
  }

  delay = time(NULL) - tstart;
  printf ("Time to open/close %d times: %lf seconds\n", ntimes, delay );
  printf ("Time per open/close: %lf seconds\n", delay/ntimes );
}
```

Exercise 30: Open for Business?

Create, compile and run Program 5.5. How fast is your computer's disk? Remember that on slower disks it can take several minutes for the program to run. If you get tired of waiting, you can stop the program by pressing Ctrl-C.

The lesson we should learn from this is that it's a good idea to avoid unnecessarily opening or closing files. If you write a simulation program like gutter.cpp in Chapter 2 and make the program write its output into a file, it's best to open the output file once, before starting any loops, and then close the file after all the loops are finished. Even though, in principle, you could open the file each time you want to write a new number, that would make your program much, much slower.

Notice that in Program 5.5 we opened the file for writing by giving a "w" as the second argument to fopen. Remember that this wipes out any already-existing file that has the same name. That's why only one small file, containing just the text "Testing", is created when the program is run. The program actually creates and overwrites this file a million times.

Accidentally overwiting an output file is a common programming error. Consider Program 5.6.

Program 5.6: overwrite-test.cpp

```
#include <stdio.h>
int main () {
  FILE *output;
  int i;

  for ( i=0; i<10; i++ ) {
    output = fopen("overwrite-test.dat","w");
    fprintf( output, "%d\n", i);
    fclose( output );
  }

}
```

This program has a loop that sets i to each value from 0 to 9 and writes that value into the output file. If the programmer ran this program he or she might be surprised to find that the output file ends up with only a single number in it: "9". That happened because the fopen and fclose statements are inside the loop, and because we gave fopen "w" (for "write") as its second argument instead of "a" (for "append"). We could fix the program by just moving fopen and fclose outside the loop, like this:

Come in, we're open!

Source: Wikimedia Commons

Program 5.7: overwrite-test.cpp, Fixed!

```
#include <stdio.h>
int main () {
  FILE *output;
  int i;

  output = fopen("overwrite-test.dat","w");
  for ( i=0; i<10; i++ ) {
    fprintf( output, "%d\n", i);
  }
  fclose( output );

}
```

Now the program's output file will look like this:

```
0
1
2
3
4
5
6
7
8
9
```

which is probably what the programmer intended.

Closing a file before the program is done with it is another common programming error. If the program above had left fclose inside the loop, then the output file would be closed after the first number was written to it. The next time the program tried writing into the file we'd get lots of ugly errors like this:

```
Error in `./overwrite-test': double free or corruption
```

This isn't very informative, but the computer is trying to tell us that we're attempting to write into a file that is no longer open.

5.8. Analyzing Other People's Data

Imagine that you're an astronomer, and you've been given the task of analyzing some data about the stars in our local neighborhood. In 1957 astronomer Wilhelm Gliese published the first edition of his list (or "catalog") of nearby stars. It contained entries for about 900 stars. By "nearby", he meant stars within about 65 light-years of Earth. Several editions later, the Gliese catalog now contains about 3,800 stars. The catalog contains information about each star's position, brightness, and color, among other things.

These stars might seem special because they're our closest neighbors. If we were ever to venture into interstellar space, these are the first places we'd visit. You've probably heard of some of them. Sirius, the "Dog Star", is the brightest star in our sky. Tau Ceti and Epsilon Eridani are two nearby Sun-like stars that figure prominently in Science Fiction.

But how close is the nearest star (other than the Sun) to us? Let's write a program to analyze some data about nearby stars and find out.

Program 5.8 reads a file containing x, y, and z coordinates (measured in parsecs[5]) for the position in space of each star. In our `readbullet` program, we analyzed some data to find the maximum value. Here we want to find the minumum value: the star that's closest to earth.

In this data's coordinate system, our Sun is at the origin. If we're given the coordinates of another star, we can find its distance from the Sun like this:

$$r = \sqrt{x^2 + y^2 + z^2}$$

where r is the distance.

Program 5.8 reads a star's coordinates from the data file `stars.dat`, then calculates the distance to that star. If that distance is less than the smallest distance we've encountered so far, the program uses it as the new value for the variable `rmin`. Compare this program with Program 5.4, which found a maximum.

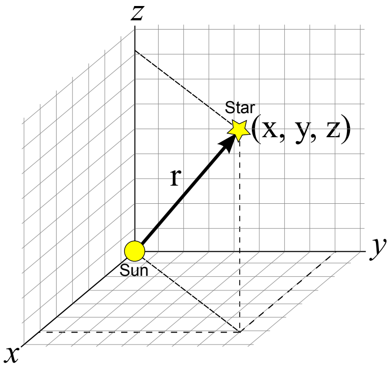


Figure 5.10: Calculating the distance from the sun to another star.
Source: Wikimedia Commons

[5] One parsec equals approximately 3.26 light years.

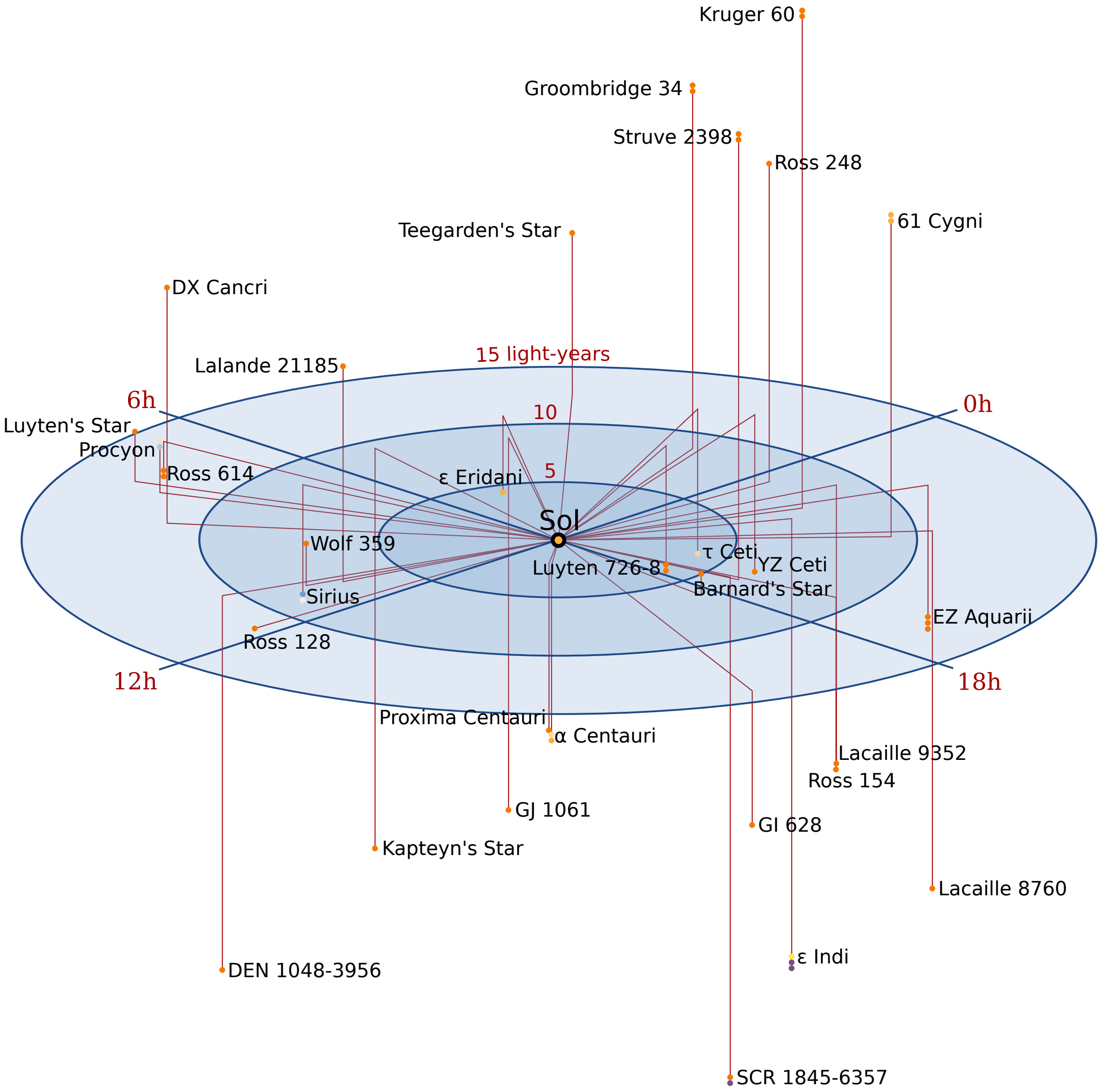


Figure 5.11: The stars in our immediate neighborhood.

Source: Wikimedia Commons

Program 5.8: stars.cpp

```
#include <stdio.h>
#include <math.h>
int main () {

  double x;
  double y;
  double z;
  double r;
  double rmin;
  int initialized = 0;

  FILE *input;

  input = fopen("stars.dat","r");

  // Read coordinates for the stars:
  while ( fscanf( input, "%lf %lf %lf", &x, &y, &z ) == 3 ) {
    r = sqrt( x*x + y*y + z*z );
    if ( initialized == 0 || r < rmin ) {
      rmin = r;
    }
    initialized = 1;
  }

  printf ( "Minimum distance is %lf parsecs\n", rmin );

  fclose( input );
}
```

Exercise 31: Seeing Stars

For this exercise you'll need a copy of the data file named `stars.dat`. You can find instructions for obtaining it in Appendix C.1 on page 621. After you have the data file, create, compile and run Program 5.8. What's the distance to the closest star in this data set? Its name is Proxima Centauri.

If you have time, start up *gnuplot* and give it the following commands (note that the last command is **s**`plot`, not `plot`):

```
set xrange [-5:5]
set yrange [-5:5]
set zrange [-5:5]
splot "stars.dat"
```

This should show you a 3-dimensional view of the stars within about 15 light years from earth. Depending on the version of *gnuplot* you're using, you may be able to grab this plot with the mouse and drag it around to rotate it.

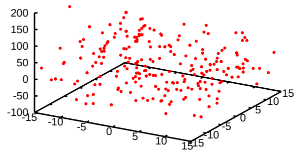


Figure 5.12: Some local stars, plotted with *gnuplot*.

5.9. Combining Files

Sometimes it's useful to be able to combine data from two or more files into one. Here are a few techniques for doing that.

Appending:

Imagine you're a teacher. You begin the semester by creating a file named `grades.dat` that will hold your students' grades. The format of the file will be one line per student, with the student's ID number at the beginning of the line, followed by a list of homework grades separated by spaces. The file might look like Figure 5.13.

```
571 95.0 89.5 100.0
292 79.5 88.0 90.0
963 82.5 87.5 95.5
894 99.0 100.0 97.5
935 88.0 89.0 91.5
616 92.0 93.5 96.0
907 100.0 99.0 95.5
288 90.0 92.0 95.0
729 88.5 92.5 95.0
710 100.0 96.5 90.0
```

Figure 5.13: Your `grades.dat` file might look like this. Each line begins with the student's ID number. After that comes a list of that student's homework grades.

After you've created this file, you find that your class is very popular but the classroom is small. You'll have to teach two groups of students at different times. To accommodate the second group of students, you create a new file `grades2.dat` with the same format as the first file.

As the semester goes along, you realize that you'd really like to have one file that contains all the grades for both sets of students. No problem! This is a programming class, so you know how to write a program for combining the two files.

You decide that you just want to append the data from `grades2.dat` onto the bottom of `grades.dat`, and then ignore `grades2.dat` from now on. To accomplish this, you write Program 5.9.

Program 5.9: append.cpp

```
#include <stdio.h>
int main () {
  FILE *file1;
  FILE *file2;
  int id;
  double h1,h2,h3;

  file1 = fopen("grades.dat","a");
  file2 = fopen("grades2.dat", "r");


  while ( fscanf( file2 , "%d %lf %lf %lf", &id, &h1, &h2, &h3 ) == 4 ) {
    fprintf ( file1 , "%d %lf %lf %lf\n", id, h1, h2, h3 );
  }

  fclose ( file1 );
  fclose ( file2 );
}
```

Open `grades.dat` for *appending* by specifying `"a"`.

Read from `file2` (`grades2.dat`).

Write to `file1` (`grades.dat`).

Program 5.9 reads each line of grades2.dat and writes it at the end of grades.dat. It's written at the end because we told fopen to open the file for appending, by specifying "a". After running this program, all of the grades would be in grades.dat.

This program shows that you can have more than one file open at a time. When we read or write, we specify which file to use by giving the appropriate file handle to fscanf or fprintf.

Concatenating:

Thinking about your class a little more, it might occur to you that it would be better to leave both grades.dat and grades2.dat as they are (since these are important student records!) and create a third, new file named homework.dat that combines the data from both the original files. You could write another program (Program 5.10) to do that.

Program 5.10: concat.cpp

```
#include <stdio.h>
int main () {
  FILE *file1;
  FILE *file2;
  FILE *homework;
  int id;
  double h1,h2,h3;

  homework = fopen("homework.dat", "w");

  file1 = fopen("grades.dat","r");
  while ( fscanf( file1, "%d %lf %lf %lf", &id, &h1, &h2, &h3 ) == 4 ) {
    fprintf ( homework, "%d %lf %lf %lf\n", id, h1, h2, h3 );
  }
  fclose ( file1 );

  file2 = fopen("grades2.dat", "r");
  while ( fscanf( file2, "%d %lf %lf %lf", &id, &h1, &h2, &h3 ) == 4 ) {
    fprintf ( homework, "%d %lf %lf %lf\n", id, h1, h2, h3 );
  }
  fclose ( file2 );

  fclose( homework );
}
```

Open the new file homework.dat for *writing* by specifying "w".

Read data from grades.dat and write it to homework.dat.

grades.dat

Now read data from grades2.dat and write it to homework.dat.

grades2.dat

As you can see, Program 5.10 creates a new file named homework.dat by giving fopen a "w". The program then has two sections: first it reads data from grades.dat and writes that data into homework.dat. Then it does the same for grades2.dat.

Merging:

All is well until the end of the semester. You've graded all of the homework assignments and put the grades into homework.dat. You've also graded some quizzes and put those grades into quizzes.dat. There were three homework assignments and two quizzes (it was a short course). Each student has one line in each file. Figure 5.14 shows what the quizzes.dat file might look like.

```
1 100.0 96.5
2 88.5 92.5
3 90.0 92.0
4 100.0 99.0
5 92.0 93.5
6 88.0 89.0
7 99.0 100.0
8 82.5 87.5
9 79.5 88.0
10 95.0 89.5
```

Figure 5.14: The quizzes.dat file might look like this, with each line containing a student's ID number and two quiz grades.

Hmmm. It would be really nice if we could combine homework.dat and quizzes.dat and create a new file that had all of each student's grades, homework and quizzes, on a single line. To do that, you could write something like Program 5.11.

Program 5.11 creates a new file named allgrades.dat that will contain one line per student, with all of that student's grades (homework and quizzes). Each line begins with the student' ID number. The new file might look like Figure 5.15.

```
1 95.0 89.5 100.0 100.0 96.5
2 79.5 88.0 90.0 88.5 92.5
3 82.5 87.5 95.5 90.0 92.0
4 99.0 100.0 97.5 100.0 99.0
5 88.0 89.0 91.5 92.0 93.5
6 92.0 93.5 96.0 88.0 89.0
7 100.0 99.0 95.5 99.0 100.0
8 90.0 92.0 95.0 82.5 87.5
9 88.5 92.5 95.0 79.5 88.0
10 100.0 96.5 90.0 95.0 89.5
```

Figure 5.15: The file allgrades.dat, produced by Program 5.11, might look like this. Each line has the students ID number, followed by three homework grades and two quiz grades.

Notice that the program reads one line from each input file each time it goes around the while loop. The fscanf statements for reading homework.dat and quizzes.dat are different, because the files have different formats. Both begin with the student ID number, but there are three homework grades and only two quizzes.

The loop stops (by using the break statement) when it reaches the end of either input file. It's important to check both files, to help us deal with mistakes we might have made when we entered the grades. What if we've left a student out of one of the files? In that case the input files wouldn't both be the same length.

Similarly, we put the student ID number into id1 when we read it from homework.dat and we put the number into id2 when we read it from quizzes.dat. If we haven't made any mistakes in creating the input files, these two ID numbers should always match. If they don't, the program gives us an error message telling us so.

Finally, once the program has successfully read a line of homework data and a line of quiz data, it writes all of the data out on a single line of the output file. Notice that the first `fprintf` statement doesn't end with a "`\n`". Instead, it ends with a space. The next `fprintf` statement picks up where the first one left off, adding more stuff to the end of the same line, and then finishing with a "`\n`".

Program 5.11: merge.cpp

```
#include <stdio.h>
int main () {
  FILE *file1;
  FILE *file2;
  FILE *combined;
  int id1, id2;
  double h1,h2,h3;
  double q1,q2;

  combined = fopen("allgrades.dat", "w");

  file1 = fopen("homework.dat","r");
  file2 = fopen("quizzes.dat", "r");

  while (1) {
      if  ( fscanf( file1, "%d %lf %lf %lf", &id1, &h1, &h2, &h3 ) != 4 ) {
       break;
      }

      if ( fscanf( file2, "%d %lf %lf", &id2, &q1, &q2 ) != 3 ) {
       break;
      }

    if ( id1 == id2 ) {
      fprintf ( combined, "%d %lf %lf %lf ", id1, h1, h2, h3 );
      fprintf ( combined, "%lf %lf\n", q1, q2);
    } else {
      printf ( "Error! IDs don't match: %d and %d\n", id1, id2);
    }
  }

  fclose ( file1 );
  fclose ( file2 );

  fclose( combined );
}
```

Output file

Input files

Read homework

Stop when we reach the end of either input file.

Read quizzes

Check to make sure the student IDs from both files match.

Write homework and quiz data on one line.

5.10. Conclusion

In this chapter we've covered the basics of reading from files and writing to files. These same techniques can be used for any numerical data that's stored in the form of multi-column, readable numbers. Programs like *gnuplot* read data files in a way very similar to this. Multi-column numerical data is very commonly used for small-to-moderate sized data sets, although sometimes the columns are separated by commas, colons or other characters besides spaces.[6]

Figure 5.16: In the days before files were stored on disks, students delivered stacks of punched cards to counters like this one in the the basement of the UNC Physics building. Computer operators loaded the stacks into readers, and the program's output was printed (sometimes hours later) and dropped by the operator into a bin, until the student came by to pick it up.

Source: UNC-Chapel Hill Computing History photo collection

[6] Large data sets are generally stored differently, in formats not readable by humans but which allow the files to be smaller, faster to read, and easier to search. We'll take a look at reading and writing this kind of files later on.

Practice Problems

1. Use *nano* to create a file named `numberlist.dat` that contains a list of numbers, one number per line, like this:

```
1.25
20.7
-4.3
123.4
```

It doesn't matter what numbers you put into the file, but put at least ten of them, some positive and some negative, and make sure they're not all integers.

Now write a program named `sumit.cpp` that reads `numberlist.dat` and tells you the sum of all the numbers in that file.

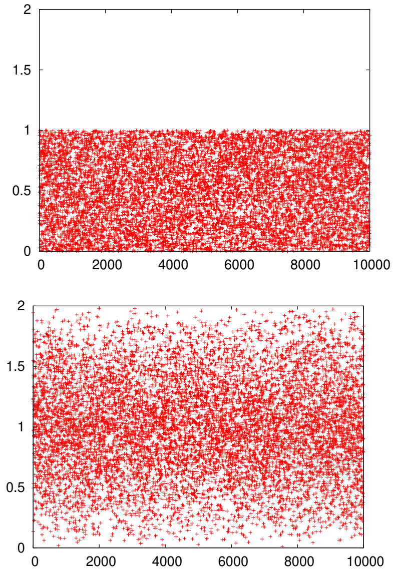


Figure 5.17: The result of `plot "randsum.dat"` (top) and `plot "randsum.dat" using 1:3` (bottom).

2. Create a program named `randsum.cpp` that writes 10,000 lines of output into a file named `randsum.dat`. Each line should contain three numbers separated by spaces: the line number (from zero to 9999); a random number between zero and one; and the sum of two random numbers each between zero and one.

Hint: As we saw in Chapter 2, you can make a random number between zero and one like this: `rand()/(1.0+RAND_MAX)`.

If you use *gnuplot* to look at the file your program makes, you should see something like Figure 5.17. Notice that the sum of two random numbers (the bottom graph) looks very different from a single random number (the top graph). The sum is still random, but it's no longer a "flat" distribution: not all numbers are equally likely. The sum is more likely to be near 1.0 than it is to be near 0.0 or 2.0.

3. Write the following two programs:

   (a) Modify Program 5.3 (the `bullet.cpp` program) so that it writes comma-separated columns into its output file, instead of space-separated columns. Run the program to generate a new `bullet.txt` output file.

   (b) Modify Program 5.4 (the `readbullet.cpp` program) so that it will read the new comma-separated data file.

4. Using *nano*, create a data file called `numbers.dat` that contains a column of at least ten integers (positive or negative), like this:

```
27
-3
189
43
-1280
```

Figure 5.18: Eastern Comma butterfly (*Polygonia comma*).
Source: Wikimedia Commons

Write a program called `readnum.cpp` that uses a "`while`" loop to read the numbers from `numbers.dat`. Make the program print out the sum of all of the numbers, the value of the largest number, and the value of the smallest number, like this:

```
Sum is -1024
Largest is 189
Smallest is -1280
```

Make sure your program does the right thing even if all the numbers are negative.

5. Using *nano*, create a file named `budget.dat` that contains three equal-length columns of numbers, like this:

```
-462.13   486.47   973.79
 755.42   843.04  -963.67
 442.58  -843.02  -462.86
-233.93  -821.67   399.59
-379.65  -556.37   837.46
  55.18  -144.93   -93.15
 533.73   804.64   -66.25
-922.12   914.68  -264.67
-600.27  -838.59   747.02
-962.97    49.96  -677.79
```

Now write a program named `budget.cpp` that reads this file and adds up the numbers in each column. The program's output should look like this:

```
Column sums are: -1774.16 -105.79 429.47
```

Note that you can limit the number of decimal places you print by using `%.2lf` instead of just `%lf`. This tells printf to print only two numbers after the decimal point.

6. Using *nano*, create the file `grades.dat` shown in Figure 5.13 on Page 166. Now write a program named `meangrade.cpp` that reads `grades.dat` and prints out a list of student IDs along with each student's average grade. Determine the average by adding up the student's grades for the three homework assignments and dividing the result by 3. The program should print "`Student ID`" and "`Mean Grade`" at the top of the output, to tell the user what the numbers mean.

Doing homework.
Source: Wikimedia Commons

7. Using *nano*, create the file `grades.dat` shown in Figure 5.13 on Page 166. Now write a program named `lowgrade.cpp` that reads `grades.dat` and prints the lowest grade for the first homework assignment, and the ID number of the student who got this grade.

Make sure your program tells the user what these numbers mean. (If there's more than one student with the lowest grade, just print the first student ID that has this grade.) Don't assume the grades will always be between zero and 100. (What if the program were given a file full of SAT scores, for example?)

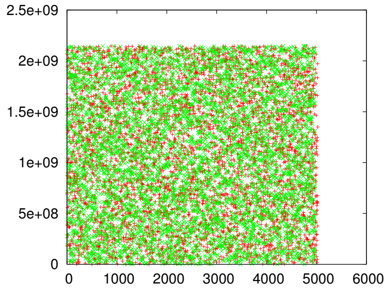


Figure 5.19: This is how the data in `odd.dat` and `even.dat` might look if plotted with *gnuplot*.

8. Write a program named `oddeven.cpp` that generates 10,000 random integers and sorts them into two files. Put the odd integers into `odd.dat` and the even integers into `even.dat`. Here are a few hints to help you:

   - You can generate a random number with the `rand` function, as we did in Chapter 2. For example:

     ```
     number = rand();
     ```

   - You can use the modulo operator, `%`, to check whether a number is positive or negative. If `number % 2` is zero, then `number` is even. Otherwise it's odd. (Look back at Chapter 4 for more information about the modulo operator.)

   You might find it interesting to look at `odd.dat` and `even.dat` with *gnuplot*. For example, if you start *gnuplot* and give it the command:

   ```
   plot "odd.dat", "even.dat"
   ```

   you should see a rectangle filled with dots of two different colors, one color for odd numbers and the other for even (see Figure 5.19). The extent of the rectangle horiontally will show you how many numbers there are of each type. About half of the numbers you generated should fall into each category, so the rectangle should go up to about 5,000. The vertical axis shows the actual numbers you generated. The height of the rectangle will depend on what kind of C compiler and computer you're using, but it should go up to some very big numbers.

Linc, Julie, and Pete: The Mod Squad

Source: Wikimedia Commons

9. Write a program named `mod3.cpp` that generates 10,000 random integers and sorts them into three files, depending on the remainder when the number is divided by 3. If the number is `n`, then the remainder is given by `n%3`, and will be either 0, 1, or 2. If the remainder is 0, write the number into a file named `zero.dat`, if it's 1, write the number into `one.dat`, and if it's 2, write the number into `two.dat`. After running the program, each of these three files should contain a single column of random numbers. Here are a few hints to help you:

   - You can generate a random number with the `rand` function, as we did in Chapter 2. For example:

```
number = rand();
```

- You'll have three files open at the same time, so you'll need three different file handles. I suggest something like this:

  ```
  FILE *zero;
  FILE *one;
  FILE *two;
  ```

  Each of these will need to be opened before you start sorting the numbers. Then you just need to pick the appropriate file for each number. For example, if `n%3` is 1 the program should do something like this:

  ```
  fprintf ( one, "%d\n", number );
  ```

After running the program, try plotting your results with *gnuplot*. The following lengthy *gnuplot* command should give you a graph similar to Figure 5.20 (type the command all on one line):

```
splot "zero.dat" using (0):($0):1,
"one.dat" using (1):($0):1,
"two.dat" using (2):($0):1
```

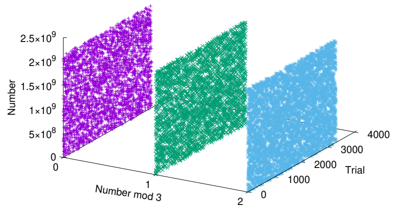


Figure 5.20: Data from the three files created by the `mod3.cpp` program.

This command plots the data from all three files in three dimensions, using one axis to represent `n%3`, another to represent the values of the random numbers, and a third axis that just counts the numbers. You should see three mor-or-less evenly filled rectangles of the same size.

10. Let's write a data-entry program that collects payment information and stores it in a file. Call the program `ledger.cpp`. The program will have an infinite "`while`" loop that asks for an account number and an amount paid into that account. If the account number is zero, the loop stops. Whenever a non-zero account number and amount are entered, those numbers are written into a file named `ledger.dat`. For example, if the user enters 100 as the account number and 1.98 as the amount, the program should write:

    ```
    100 1.98
    ```

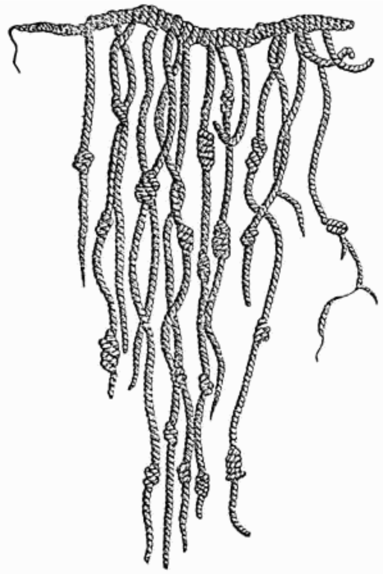
    into the `ledger.dat` file.

    While the program is collecting amounts, it should also sum them up. At the end of the program it should write the sum on the screen (not in `ledger.dat`) in a form like this:

    ```
    Total amount entered was 293.600000
    ```

    (Don't worry about how many decimal places the numbers show, as long as it's at least two.)

    Hint: Remember that you can use a `break` statement to stop a loop.



The Incas of South America used knotted strings called quipus to record financial transactions. Alex Bellos describes how quipus work in a Numberphile video.

Source: *Wikimedia Commons*

11. Although the surface of the sun is very hot (about 6,000 kelvin), there are slightly cooler spots that look dark when the sun is viewed through a filter that blocks most of its light. Sunspots are temporary, and their number varies over time in an 11-year cycle.

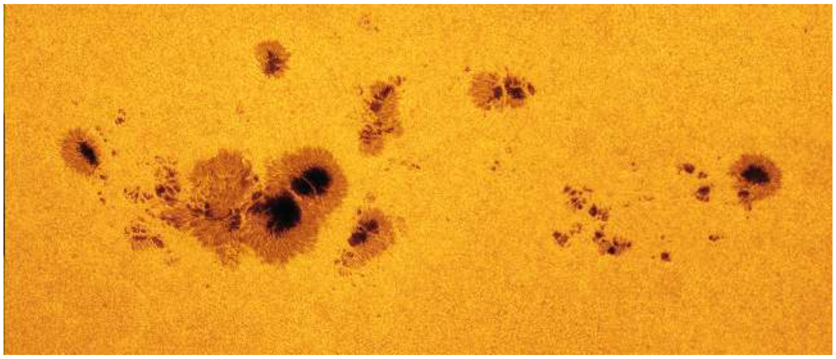


Figure 5.21: A large group of sunspots on July 7, 2012. The largest spot in this photo is about 11 times as wide as the whole Earth.

Source: Wikimedia Commons

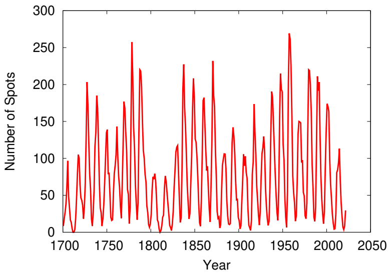


Figure 5.22: The number of sunspots varies in an 11-year cycle. This graph shows the number of sunspots each year from 1700 to 2021.

As you can see from Figure 5.22, each cycle has a different maximum number of sunspots. Let's write a program named `sunspots.cpp` that reads some sunspot data and finds the year with the highest number of sunspots.

First you'll need to fetch the data. You can do that by typing either:

```
wget https://www.sidc.be/silso/DATA/SN_y_tot_V2.0.txt
```

or

```
curl -L -O https://www.sidc.be/silso/DATA/SN_y_tot_V2.0.txt
```

whichever works on your computer. This file contains sunspot data for all the years between 1700 and 2021. Each line has four columns: The year, the average number of sunspots per day, an estimate of the error in this number, and the number of observations[7]. The downloaded file will be named `SN_y_tot_V2.0.txt`.

Your program should read this file and tell the user which year had the most sunspots per day, and how many sunspots that was. See Program 5.4 on page 159 for some hints about how to write your program.

12. Modify Program 5.3 (the "`bullet`" program) so that it uses a "`do-while`" loop to track the bullet until it reaches the ground. (See the gray box after `bullet` program for information about how to do this.) Make the program write out how long (in seconds) it takes the bullet to reach the ground. Call the new program `bullettimer.cpp`.

[7] The last two columns contain "-1" for the early years, since this information isn't available.

13. In 1912 the ocean liner *Titanic* hit an iceberg during the ship's maiden voyage across the Atlantic. About 1,500 passengers and crew died as a result.

BUSINESS PROPERTY

New York American

EDITION FOR GREATER NEW YORK.

TUESDAY, APRIL 16, 1912. 16 PAGES

PRICE ONE CENT

J. J. ASTOR LOST ON TITANIC

1.500 TO 1.800 DEAD

John Jacob Astor was among the passengers who went down with the ship, according to a wireless dispatch received by Bradstreets last night from the liner Olympic. Mrs. Astor was saved and is being brought to shore by the Carpathia.

The Wireless Operator at Cape Race, Newfoundland, Flashes: "Eighteen Hundred Lives Have Been Lost in the Wreck of the Titanic."

Figure 5.23: Headlines announce the disaster.

Source: Wikimedia Commons

Let's write a program to analyze a file full of data about passengers on the *Titanic*. First download the file by doing either:

```
wget http://jse.amstat.org/datasets/titanic.dat.txt
```

or

```
curl -L -O http://jse.amstat.org/datasets/titanic.dat.txt
```

If you use *nano* to look inside this file you'll see that it contains four columns of numbers. Each line of the file describes one person. In the first column is a number that tells whether the person was a crew member, or a first-, second-, or third-class passenger (0=crew, 1=first, 2=second, 3=third). The second column indicates age: (1=adult, 0=child). The third column divides people into two sexes: (1=male, 0=female). The final column indicates whether the person survived or not: (1=yes, 0=no).

Write a program named `titanic.cpp` that reads `titanic.dat.txt` and tells the user how many people survived in each of these four groups: Crew members, First Class passengers, Second Class, and Third Class.

Margaret Brown, known later as "the unsinkable Molly Brown", was a survivor of the *Titanic* disaster. She went on to be a world-renowned advocate for womens' rights, workers' rights, education, and many other causes.

Source: Wikimedia Commons

6. Using Arrays

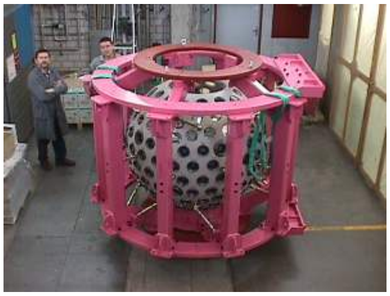
Galileo used his pulse to measure how long it took a ball to reach several marked locations while rolling down a ramp. This experiment established that the distance traveled is proportional to the square of the elapsed time, no matter how much the ball weighs.

Source: Wikimedia Commons

6.1. Introduction

Scientists often make groups of similar measurements under different conditions. We might measure the temperature of a metal bar at several different points along its length for example, or measure the velocity of a dropped ball at several times during its fall. A modern high-energy physics experiment might record the amount of energy deposited in each of hundreds of detectors every time an interesting event is seen.

Programs that analyze data need to store such measurements in variables. We could define one variable for each measurement, giving them names like `t1`, `t2`, `t3` and so forth, but that would be awkward if there were hundreds of measurements. For example, imagine adding them all up: we'd need to write an expression like `t1 + t2 + t3 + ...`, and we'd need to remember to change it if we added or removed any measurements the next time we used the program.



This detector assembly consists of 240 cesium iodide crystals. Each of them measures the energy of particles that pass through the crystal.

Source: PiBeta Collaboration

C provides us with an easier way of storing a group of related values. An "array" is a numbered list of boxes in the computer's memory. The array as a whole has a single name, and individual boxes can be referred to by number. In this chapter we'll see how to create and use arrays.

6.2. A Coal Train

Imagine that you're in charge of a rail system carrying coal. Each train has some number of coal cars, and each car can carry some amount of coal up to a maximum capacity. You'd like to keep track of how much coal is in each car, but you're also interested in the total amount of coal that the train is hauling. How might you store all of those numbers in a program?

A coal train in eastern Wyoming.

Source: Wikimedia Commons

Program 6.1 uses an *array* to store the weight of coal in each car. The array is defined by the statement:

```
double carweight[100];
```

This statement defines an array of one hundred elements[1], each capable of storing a floating-point number. The elements are numbered from zero to 99.

We can refer to a particular element of the array by giving its number. For example, if we wanted to print out the value in element number 27 of the array, we could write `printf("%lf",carweight[27]);`. It's very important to remember that the last element in the array is `carweight[99]`, *not* `carweight[100]`. When we define the array, we say how many elements are in it, but the elements are numbered starting with zero, so the last element will always have a number that's *one less than* the total number of elements.[2]

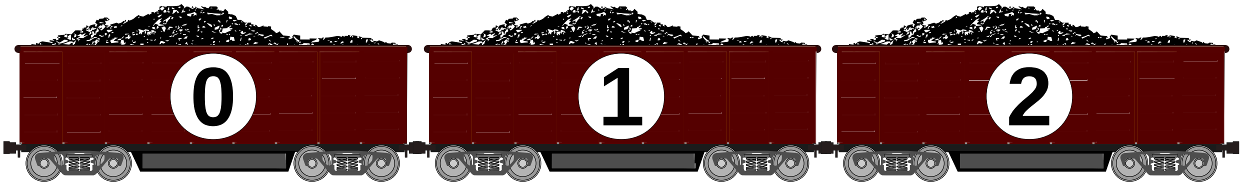


Figure 6.1: The first element of an array is number *zero*.
Source: Openclipart.org

The first loop in Program 6.1 puts a random weight of coal into each of the cars. The weights vary between 50 and 100 tons. In a real-world program, these weights probably wouldn't be random. They might be read out of a file, or they might be read from some kind of device that measures each car's weight as it goes by. This is just an example, though, so we'll use random numbers.[3] Notice that we can set the value of one of the array's elements by referring to it by number.

The program's second loop just prints out the weight of each car in a nice, readable format. Notice that the value of *i* in both loops runs from zero to 99, since the loop starts at zero and continues for as long as *i* is less than 100 (`i<100`).

Program 6.1 also tells us the total amount of coal the train is carrying. The variable `sum` starts with a value of zero, then has the weight of

[1] Each "element" of an array is just a storage box for holding something.

[2] Programmers often refer to an element's number as its "index". Array indices are like the subscripts we use in mathematics when we write an expression like X_i. The index must be an integer, since it just counts the number of elements.

[3] Since we don't use the *srand* function to change the random number generator's seed, the program will always give us the same set of "random" numbers. (See Chapter 2.)

each car added to it. At the end of the program, the total weight of all cars is printed.

Program 6.1: coal.cpp

```
#include <stdio.h>
#include <stdlib.h>
int main () {
  double carweight[100];
  double w;
  double sum = 0.0;
  int i;

  for ( i=0; i<100; i++ ) {
    w = 50.0 + 50.0 * rand()/(1.0 + RAND_MAX);
    carweight[i] = w;
  }

  for ( i=0; i<100; i++ ) {
    printf ( "Car %d carries %lf tons\n", i, carweight[i] );
    sum += carweight[i];
  }

  printf( "The total weight of coal is %lf tons.\n", sum );
}
```

Exercise 32: "I think I can..."

Create, compile and run Program 6.1. Notice that the car numbers (the array indices) start at zero and end at 99.

Think about how you'd need to change the program to accommodate 200 cars instead of 100. What would be the index of the last car then?

6.3. How Arrays Are Stored

In our programs, a variable is just a temporary storage location in the computer's memory that has a name attached to it. The size of this storage location depends on the type of data we want to put into it. Just as a violin case is different from a trombone case, the box of memory reserved for an `int` variable will be different from the box reserved for a `double` variable.

Figure 6.2 shows how a group of variables might be placed in the computer's memory. Note that `int` and `double` variables require different-sized storage boxes. The data inside these boxes is also organized differently. Because of this, even though `int` data would fit into the space reserved for a `double`, the data would appear garbled when your program tried to read it, because the program would try to interpret these bits as a floating-point number. You might be able to squeeze a violin into a trombone case, but imagine trying to play the violin by blowing into it like a trombone!

The great jazz violinist Stephane Grappelli.

Source: *Wikimedia Commons*

"Trombone Shorty" (aka Troy Andrews) began playing the trombone before the age of six, when he was so small he had to use his feet to reach the low notes.

Source: *Wikimedia Commons*

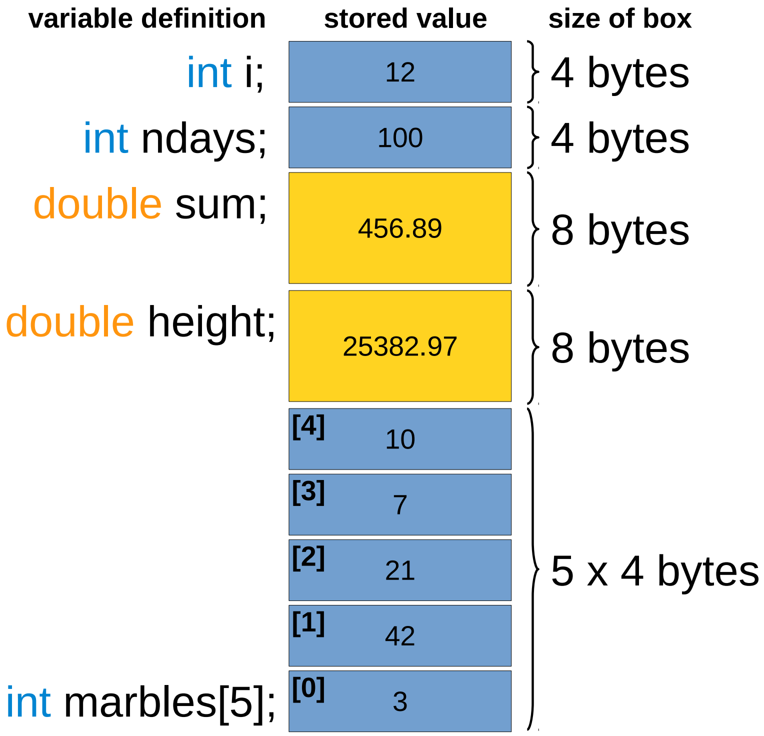


Figure 6.2: How a group of variables might be arranged in the computer's memory. The actual size of `double` or `int` variables may differ depending on the type of computer, operating system, or C compiler. The values shown here are typical, though.

Figure 6.2 also shows how an array is stored. In the figure, a five-element `int` array named `marbles` is defined. Imagine that it records the number of marbles in each of five bags. As you can see, this array takes up the same amount of storage space as five regular `int` variables.

It's important to remember that each element of an array takes up just as much memory as a separate variable of that type. So, if we define a large array with thousands of elements, we may run into the limits of the computer's memory.

The elements of an array are always stored one after another in the computer's memory. You could think of them as a stack of shoe boxes. In fact, when you ask the computer to find, say, `marbles[3]` it finds the memory address of the first element of `marbles` and then just skips forward by a distance equal to three times the size of a single `int` variable.

All of the elements of an array must have the same type, but this can be `int`, `double`, or any other type that C provides. In our train example, we defined an array of `double` elements called `carweight`. In Figure 6.2 we define an array of `int` elements called `marbles`.

If you have a small array, you might find it useful to set the initial values of the elements when you define the array. For the `marbles` array, for example, we could define and initialize the array by saying `int marbles[5] = {3,42,21,7,10};` The list of numbers in curly brackets will be put into elements zero through five of the array.

But what about...?

Is there a way to find out how much storage space is needed for a type of variable? Yes! You can use the `sizeof` function to find the size of a type, or of a particular variable.

Take a look at this example:

```
#include <stdio.h>
int main () {
  int i;
  double x;

  printf ("Size of int is %d bytes.\n", (int)sizeof( int ) );
  printf ("Size of double is %d bytes.\n", (int)sizeof( double ) );

  printf ("Size of i is %d bytes.\n", (int)sizeof( i ) );
  printf ("Size of x is %d bytes.\n", (int)sizeof( x ) );
}
```

If you ran this program, the output would look something like this:

```
Size of int is 4 bytes.
Size of double is 8 bytes.
Size of i is 4 bytes.
Size of x is 8 bytes.
```

The sizes may be different on your computer, but you can always use `sizeof` to find them if you need them. (Note that we force the value of `sizeof` to be an `int` by putting "`(int)`" in front of it. This is necessary on some computers because the value returned by `sizeof` isn't strictly an `int`.)

6.4. Selecting Array Elements

Let's get back to work on our coal-hauling business. As our train is travelling across the country, we might want to look up the weight of a particular car. Maybe we have a customer in Schenectady who wants at least 85 tons of coal. Will the last car in the train be full enough, or do we need to pick another one?

Program 6.2 adds another section to our earlier program. Now, after the program has listed the weights of all the cars and told us the total weight, it begins waiting for us to enter a car number, and will tell us how much coal is in that particular car.

Program 6.2: coal.cpp, Version 2

```
#include <stdio.h>
#include <stdlib.h>
int main () {
  double carweight[100];
  double w;
  double sum = 0.0;
  int i;
  int carno;

  for ( i=0; i<100; i++ ) {
    w = 50.0 + 50.0 * rand()/(1.0 + RAND_MAX);
    carweight[i] = w;
  }

  for ( i=0; i<100; i++ ) {
    printf ( "Car %d carries %lf tons\n", i, carweight[i] );
    sum += carweight[i];
  }

  printf ("The total weight of coal is %lf tons.\n", sum );

  while (1) {
    printf ( "Enter car number (-1 to quit): " );
    scanf ( "%d", &carno );
    if ( carno < 0 ) {
      break;
    }
    printf ( "Car number %d carries %lf tons.\n", carno, carweight[carno] );
  }

}
```

Exercise 33: Runaway Train!

Create, compile and run Program 6.2. Try entering some numbers between zero and ninety-nine. Enter -1 to stop. Do the results look reasonable?

Now try entering `1000` and `1000000`. These values are clearly beyond the end of the train. What does the program do?

Look out! Arrays give us a lot of new abilities, but they also introduce a whole trainload of potential pitfalls to beware of.

Source: Wikimedia Commons

6.5. Checking Array Index Values

Many programs can run simultaneously on a modern computer. To keep programs from interfering with each other, the computer assigns a separate chunk of memory to each program. A program is only allowed to use the memory that belongs to it.

When your coal train program starts running, the computer reserves enough memory space to hold all of the variables you've defined, including the 100 elements of the `carweight` array.[4] However, as demonstrated in the exercise above, the computer *doesn't check your array indices* to make sure they stay within the bounds of the array. This can cause problems if you're not careful when writing your program.

Take a look again at Figure 6.2. If we asked the program to print out the value of `marbles[14]` the computer would happily skip forward 14 × 4 bytes from the beggining of the `marbles` array, and try to read whatever was at that memory location.

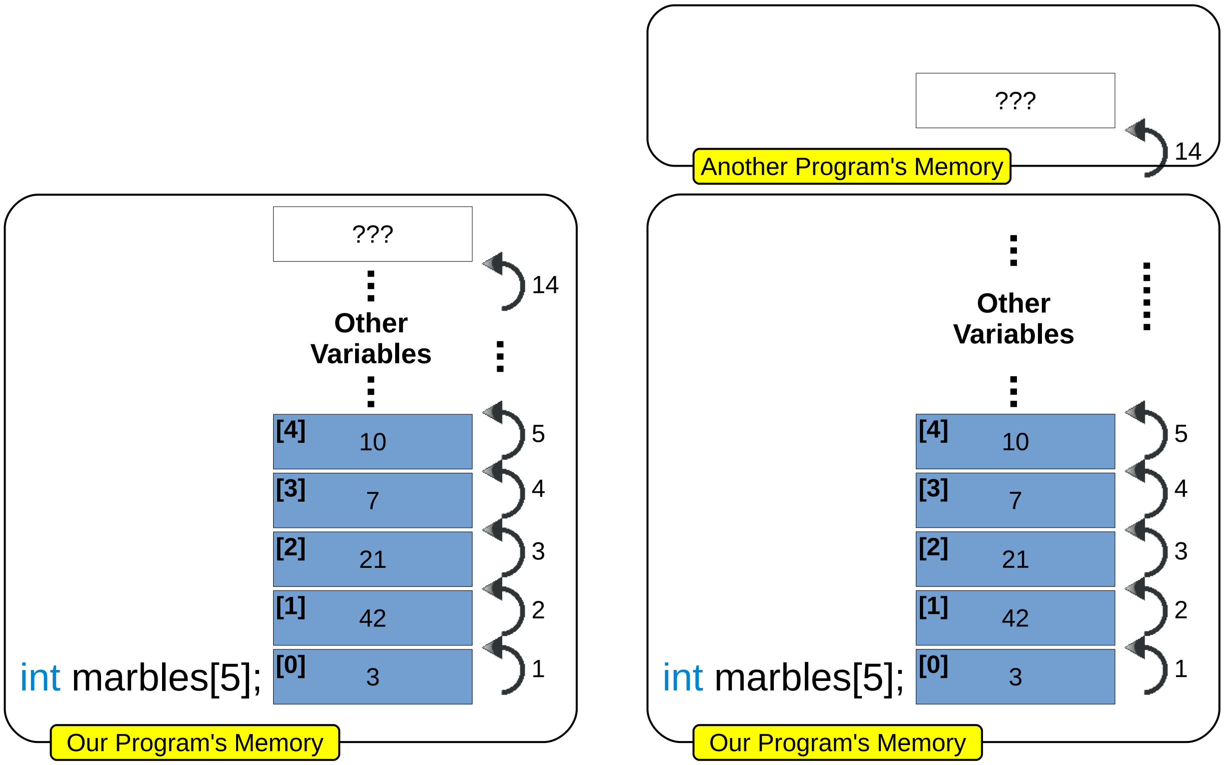
If that part of memory is in the chunk belonging to our program, then the program will be able to successfully read whatever unpredictable value happens to be stored there (see Figure 6.3*a*). If this part of the computer's memory doesn't belong to our program, then the program will crash (see Figure 6.3*b*). Usually, a crash like this generates an error message that says "Segmentation fault". This means that the program has tried to do something in a segment of the computer's memory that doesn't belong to it.

This might be an even worse problem if we tried to *change* the value of `marbles[14]`. In that case, if the program didn't crash, we'd be unexpectedly *modifying* the value of some completely different variable in our program.

If Jesse James were alive today he might have robbed computers instead of trains. Don't give Bad Guys a break! Check to make sure your array indices don't stray outside your arrays.

Source: Wikimedia Commons

[4] The memory reserved in this way is called "the stack", because it's like a stack of storage boxes, as illustrated in Figure 6.2.



(a) Reading beyond the end of an array, but still staying within the memory allocated to this program. This will succeed, but the number you get will likely be nonsense.

(b) Attempting to read outside the memory allocated for this program. This will fail and cause the program to crash.

Figure 6.3: Reading past the end of an array will give unexpected results.

It's up to the programmer to prevent these problems. In Program 6.2 for example, we could add an "`if`" statement to check to see if the number entered is between zero and ninety-nine, and tell the user to pick another number if it's not.

Reading or writing past the end of an array is one of the most common programming mistakes. It has led to many bugs in many programs, including some serious security bugs. Imagine what could happen if a banking program accidentally allowed users to change the value of any variable by entering, say, a very large account number! Bad Guys routinely look for bugs like this, and try to exploit them.

Let's move away from the hot, dirty coal industry for a little while now, and visit the cool, clean world of mathematics.

6.6. The Sieve of Eratosthenes

Prime numbers have fascinated mathematicians since ancient times. You'll recall that a prime number is a whole number that can only be divided evenly by itself and one. The first five prime numbers are 2, 3, 5, 7 and 11. (the number 1 isn't considered to be a prime.) Numbers that aren't prime are called *composite* numbers.

Early on, the Greek mathematician Euclid proved that there are infinitely many prime numbers. There doesn't, however, seem to be any simple rule for predicting them all. You just have to find them by searching.

Euclid, who lived around 300 BCE, is best known as the father of geometry.

Source: Wikimedia Commons

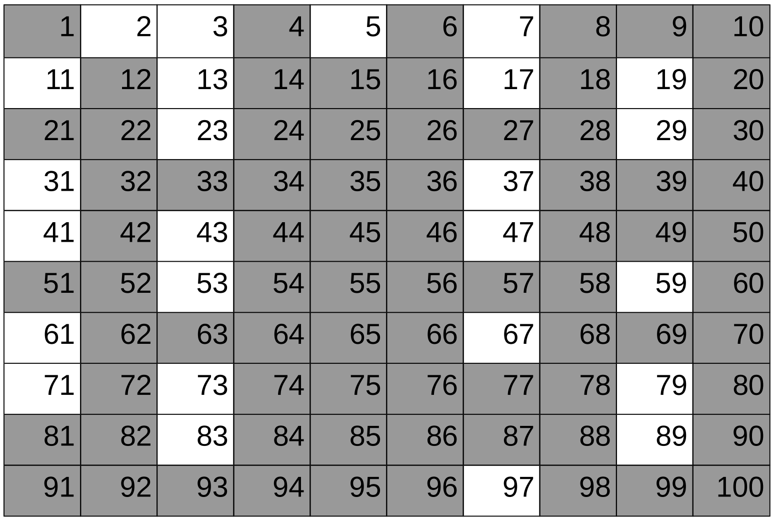
Another Greek mathematician, Eratosthenes, described a straightforward procedure for searching for prime numbers. Today we call his technique "the Sieve of Eratosthenes". It finds primes by a process of elimination. First, write down all numbers in a range, and then mark out the ones that aren't prime. Anything left over (the "holes" in the sieve) is prime. But how to you know which numbers to eliminate?

Here's how it works: Write down all of the numbers from one to N, where N is the highest number you want to test. Then mark out all the multiples of 2 (4, 6, 8, . . .). We know that none of these numbers can possibly be prime, since they can be divided evenly by 2. After that, mark out all of the multiples of 3 for the same reason, and so on. When you've gone through all of the numbers, anything that hasn't been marked out isn't a multiple of anything but 1 and itself, so it's a prime number. Figure 6.4 shows what it might look like after you'd

Eratosthenes, born around 276 BCE, is perhaps best remembered for his remarkably accurate determination of the radius of the earth. (No, the ancient Greeks didn't think the earth was flat!)

Source: Wikimedia Commons

done this for the numbers 1 to 100.



1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Figure 6.4: White squares show the prime numbers between 1 and 100. Gray squares are numbers that have been marked out by the sieve process.

Program 6.3 uses Eratosthenes' technique to find all of the prime numbers smaller than 100,000. In terms of the description above, the program sets N equal to 100,000. It begins by defining an $N + 1$ element array named `isprime` that will hold the "prime status" of each number. If the number i is prime, then `isprime[i]` will be equal to 1. Otherwise, this value will be zero.

Why does the array need $N + 1$ elements? Remember that the last element of a 100-element array is number 99, not 100, since the first element is number zero. If we want the last element of `isprime` to be number N, then the array needs to have $N + 1$ elements.

Notice that, instead of writing "`int isprime[100001];`" we've defined a variable, `N`, that says how many elements are in our array. The size of an array can't be changed once it's defined, though, so it's a good idea to mark a variable used this way as a "constant". By putting the word `const` in front of a variable definition, you tell the compiler that the value of this variable will never change.[5] If you try to change the variable's value somewhere later in the program, the compiler will give you an error message and refuse to compile the program.

Program 6.3 assumes that all of the numbers are prime unless proven otherwise. The first "`for`" loop initializes all of the elements of `isprime` to a value of 1.

The next "`for`" loop begins with 2, and goes through all of the multiples of 2 that are smaller than N. For each of these multiples, the program

[5] "constant variable?" Isn't that an oxymoron?

sets the corresponding element of `isprime` to a value of zero, thus flagging this number as a non-prime. The program then works its way through multiples of other numbers, up to N.

When it's done, anything that still has an `isprime` value of 1 is really a prime. The program prints out these numbers, and a count of how many primes were found.

The study of integers is an important part of the branch of mathematics called "number theory". Mathematician Leopold Kronecker famously said "God made the integers, all else is the work of man."

Source: Wikimedia Commons

You can probably think of some shortcuts we could have taken to make our program run faster. For one thing, if you've worked partway through the list and come to, say, 31, you know without going any farther that 31 is prime, since only smaller numbers could possibly be its factors. For another thing, it turns out that you only need to look for multiples of prime numbers. All the multiples of 4, for example, will already have been marked out, since they're also multiples of 2, and all multiples of 6 are also multiples of 2 and 3, which have already been marked out. Finally, we only need to test multiples of numbers smaller than $\sqrt{N}$. Any larger, non-prime numbers smaller than N must be a multiple of one of these.

To keep the program simple, Program 6.3 doesn't use these shortcuts. It trades speed for simplicity. This is a choice you'll often have to make as a programmer. Is a simple program fast enough? If I make the program more complicated in order to gain some speed, will I be more likely to do something wrong?

Exercise 34: Prime Time

Create, compile and run Program 6.3. How many primes does it find? Think about what problems you might run into if you tried to use this program to find even larger prime numbers.

N	Number of Primes
10	4
100	25
1,000	168
10,000	1,229
100,000	9,592
1,000,000	78,498
10,000,000	664,579
100,000,000	5,761,455
1,000,000,000	50,847,534
10,000,000,000	455,052,511
100,000,000,000	4,118,054,813
1,000,000,000,000	37,607,912,018
10,000,000,000,000	346,065,536,839

Figure 6.5: The number of primes less than N, for various values of N.

Source: https://primes.utm.edu/howmany.html

Program 6.3: sieve.cpp

```
#include <stdio.h>
int main () {
  const int N = 1e+5;
  int isprime[N+1];  // Why N+1? Number of elements, INCLUDING ZERO!
  int i;
  int multiple;
  int nprimes = 0;

  // Start by assuming everything is prime:
  for ( i=0; i<=N; i++ ) {
    isprime[i] = 1;
  }

  // Mark the non-primes:
  for ( i=2; i<=N; i++ ) { // Don't want to include multiples of 1!
    multiple = i+i; // First multiple of i
    while ( multiple <= N ) {
      isprime[multiple] = 0;
      multiple += i;
    }
  }

  // Print out what's left:
  for ( i=2; i<=N; i++ ) { // Why 2? Zero and 1 aren't prime by definition.
    if ( isprime[i] == 1 ) {
      printf ( "%d\n", i );
      nprimes++;
    }
  }

  printf ( "Total number of primes below %d is %d\n", N, nprimes );
}
```

6.7. Reading Array Elements

Because C doesn't prevent us from going past the end of an array (see Section 6.5 above) we need to be careful when we read data from a user or from a file and put it into an array. Take a look at Program 6.4, for example. This program defines a 5-element array named `marbles`, and asks the user to enter numbers into it, one element at a time. The numbers are then printed out in reverse order.

Figure 6.6: The airplane image on this 1918 "Inverted Jenny" stamp was accidentally printed upside-down. Only 100 such stamps are known to have been printed, making them very valuable to collectors. In 2007 one of these stamps was sold for almost $1,000,000.

Source: *Wikimedia Commons*

Notice that the program uses "`for`" loops that systematically go through the array's indices, from zero to 4. (Remember that the last element of a 5-element array is numbered 4, since the first element's number is zero.) Also notice that we put the array element into the `scanf` statement in just the same way that we'd put a non-array variable. In particular, we still need to put an ampersand in front of it.

After reading the numbers, the program prints them out in reverse order. It does this by starting with the last array element and working backwards through the array. We could have done this by saying "`for ( i=4; i>=0; i-- )`", but we've chosen to do it a different way. The program uses the same kind of "`for`" loop that it used when reading the numbers, but instead of printing `marbles[i]` it prints `marbles[4-i]`. Since `i` starts at zero and goes to 4, the value of `4-i` starts at 4 and goes to zero.

Program 6.4: reverse.cpp

```
#include <stdio.h>
int main () {
  int marbles[5];
  int i;

  for ( i=0; i<5; i++ ) {
    printf ( "Enter a number: " );
    scanf ( "%d", &marbles[i] );
  }

  printf ( "Numbers in reverse order:\n" );
  for ( i=0; i<5; i++ ) {
    printf ( "%d\n", marbles[4-i] );
  }
}
```

If you run Program 6.4 it might look like this:

```
./reverse
Enter a number: 2
Enter a number: 7
Enter a number: 5
Enter a number: 1
Enter a number: 9
Numbers in reverse order:
9
1
5
7
2
```

Exercise 35: Doing Flips

Create, compile and run Program 6.4. Does it work as expected?

Array indices give us a way to uniquely identify each element of an array, but they can also provide information about relationships between elements. For example, they tell us the order of the cars in our coal train, or the order of the numbers we entered in Program 6.4.

6.8. Sorting the Elements of an Array

We sometimes want to sort the elements of an array based on the values they contain. In our coal train example, for instance, we might want to put the heaviest cars at the back of the train, and the lightest at the front.

One of the simplest (but, unfortunately, slowest) ways to sort things is called a "bubble sort". Let's write a program that uses a bubble sort to arrange the cars of our train from lightest to heaviest.

A bubble sort works by comparing the values in two neighboring elements of an array. If the two values are in the proper order already (light car in front of heavy car, in our train example), they're left alone. Otherwise the two values are swapped to put them into the right order. We go through each pair of elements in the array, from first to last, swapping values when necessary. Then we do this again and again, until no more values need to be swapped. At that point, the array has been completely sorted. Figure 6.7 shows what the first pass might do to the values in our marbles array.

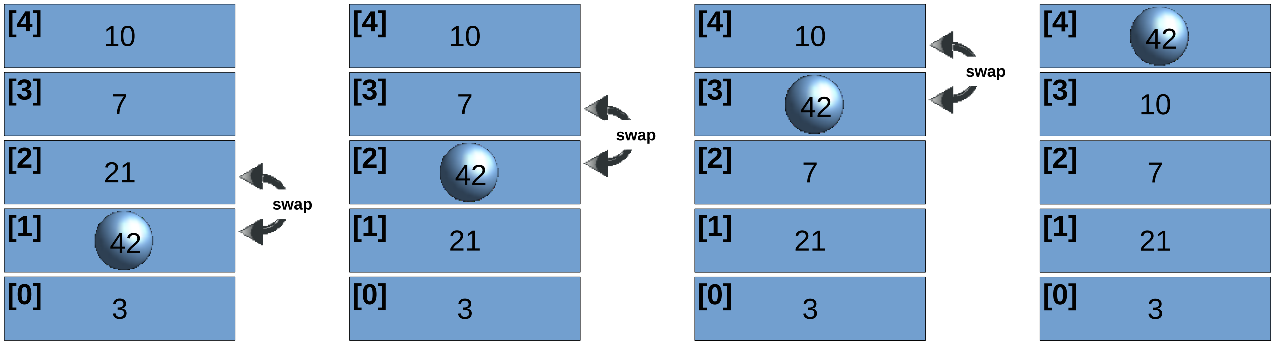


Figure 6.7: A bubble sort works its way through this array from bottom to top, comparing neighboring numbers and swapping them where necessary. When we get to the top of the array, we see that the largest number has "bubbled up". We could then start back at the bottom and repeat this procedure until all of the numbers had been sorted.

To write a program that does this, we'll first need to think about how to swap the values of two elements of an array. We can't just copy, say, `marbles[1]` into `marbles[2]`. If we did, we'd have two copies of the value in `marbles[1]`, and would have lost the value of `marbles[2]` completely! To swap values in a program, we'll generally need to have a temporary storage place to put one of the values while we're moving things around. This is illustrated in Figure 6.8.

Once we know how to swap the values in two elements, we're ready to write our bubble sort program. Program 6.5 is the result. The middle of the program is two nested loops.

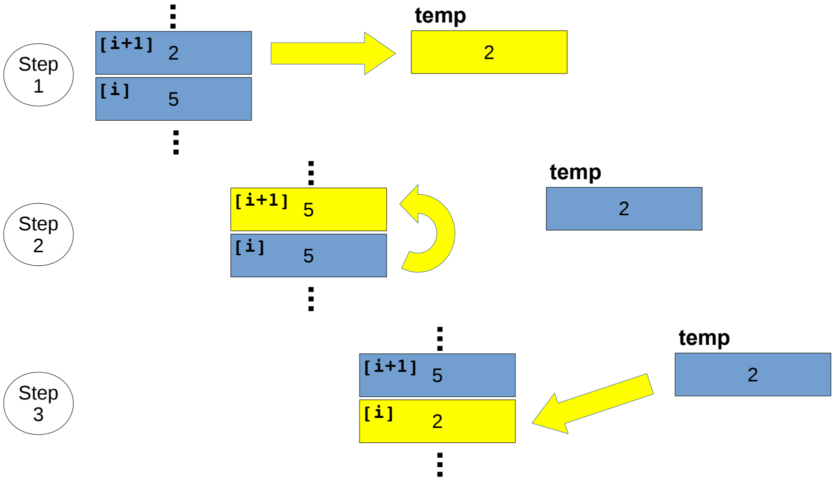


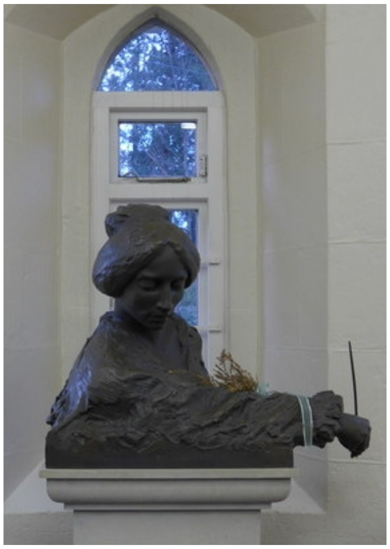
Figure 6.8: Swapping two values usually requires a temporary storage location. This illustration shows to swap the values in two adjacent elements of the `marbles` array. We use a variable called `temp` as a place to park one of the values while we're moving things around.

The inner loop is a "`for`" loop. This loop goes through each pair of array elements, starting with elements zero and one, then going to one and two, two and three, and so forth. The last pair will be 98 and 99, since the last element is number 99. The loop's counter variable, `i`, identifies the first member of each pair. The second member is `i+1`. The loop stops when `i` is equal to 98 and `i+1` is equal to 99 (the last element of the array).

The variable `temp` is a temporary storage location for use while swapping values, as shown in Figure 6.8. The variable `nswapped` keeps track of how many pairs needed to be swapped. before we begin each pass through the elements, `nswapped` is reset to zero.

The outer "`do-while`" loop repeats the inner loop until there are no more pairs that need swapping, indicated by a value of zero for `nswapped`.

A bubble sort is a simple sorting algorithm. An "algorithm" is just a recipe for doing something. Bubble sorts are easy to write, but there are much faster sorting algorithms. We'll look at one of these called "qsort" in a later chapter.



"A little later, remembering man's earthly origin, 'dust thou art and to dust thou shalt return,' they liked to fancy themselves bubbles of earth. When alone in the fields, with no one to see them, they would hop, skip and jump, touching the ground as lightly as possible and crying 'We are bubbles of earth! Bubbles of earth! Bubbles of earth!"' —Flora Thompson, in *Lark Rise* (1939)

Source: ©Basher Eyre and licensed for reuse under this Creative Commons license

Program 6.5: bubble.cpp

```
#include <stdio.h>
#include <stdlib.h>
int main () {
  double carweight[100];
  double w;
  int i;
  double temp;
  int nswapped;

  for ( i=0; i<100; i++ ) {
    w = 50.0 + 50.0 * rand()/(1.0 + RAND_MAX);
    carweight[i] = w;
  }

  do {
    nswapped = 0;
    for ( i=0; i<99; i++ ) {  // Note: omit last element!
      if ( carweight[i] > carweight[i+1] ) {
        temp = carweight[i];
        carweight[i] = carweight[i+1];
        carweight[i+1] = temp;
        nswapped++;
      }
    }
  } while (nswapped > 0);

  for ( i=0; i<100; i++ ) {
    printf ("Car %d carries %lf tons.\n", i, carweight[i] );
  }
}
```

But what about...?

Where does the word “algorithm” come from anyway? Surprisingly, it has nothing to do with Al Gore. Instead, it’s a variation on the name of Muhammed ibn Musa al-Kwarizmi. al-Kwarizmi was an 8th-Century Persian mathematician who adopted a revolutionary new Indian method for writing numbers: the decimal number system we still use today. Before decimal numbers, arithmetic was a tedious process only known to specialists. Decimal numbers suddenly made arithmetic accessible to the masses.

al-Kwarizmi’s writings, translated into Latin, brought the new number system to Europe, along with other insights into mathematics. The word “algebra” comes from the Arabic word *al-jabr*, meaning “make whole”, used in the title of one of al-Kwarizmi’s books: *al-Kitab al-mukhtasar fi hisab al-jabr wal-muqabala* (*The Compendious Book on Calculation by Completion and Balancing*).

al-Kwarizmi’s mathematical writings were so influential that his name, transmogrified into “algorithm”, became a shorthand for calculation in general.

A page from one of al-Kwarizmi’s books.

Source: Wikimedia Commons

6.9. Fun with Metronomes

A metronome is a device that clicks in a regular rhythm. Music students sometimes use them while practicing. These devices have a straight, weighted arm that swings back and forth. Imagine that you have several metronomes sitting at various widely-separated places in a room. The metronomes are all ticking at the same rate, but the arm of each metronome has been set in motion at a different time. It might look like the top half of Figure 6.9. At any given time, the arms of the metronomes are in different places. We say that the metronomes are "out of phase", and they would stay that way for as long as we could tolerate their maddening ticking!

Now imagine we take the metronomes and put them side-by-side on a wobbly table[6]. Again, we start their arms moving at different times, so they’re out of phase. But now we’d find that, over time, the metronomes begin to synchronize, until they are eventually "in phase" with each other, with all the arms at the same position at any given time, like the bottom half of Figure 6.9.

[6] This example was inspired by Matt Parker’s video on this topic: https://www.youtube.com/watch?v=J4PO7NbdKXg

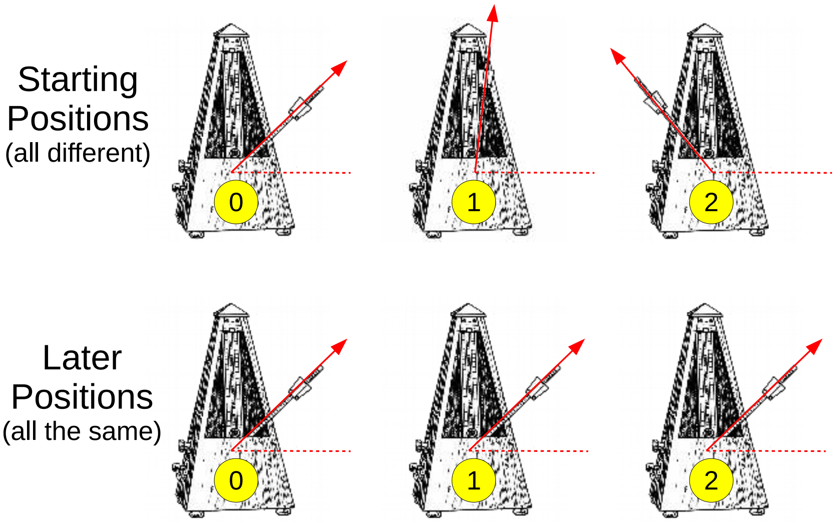


Figure 6.9: Even though the metronomes' arms might start out in different places, they can influence each other over time.

What's happening here is that the wobbly table lets the metronomes jiggle each other a little bit. We say that they're now "coupled", whereas they were "uncoupled" when they were spread out around the room. Over time, the coupling between the metronomes tends to bring them into phase with one another.

In the 1970s Yoshiki Kuramoto developed a simple mathematical model[7] that describes how the metronomes' motion evolves from out-of-phase to in-phase. Let's write a program that uses Kuramoto's model to simulate the behavior of a set of metronomes on a wobbly table.

We're going to need to keep track of each metronome's arm as it oscillates back and forth. Figure 6.10 shows the motion of a single metronome. The vertical axis shows the position of its arm, where 1 means all the way to the right, and -1 means all the way to the left. As time passes, the arm oscillates between these two extremes in a sine wave.

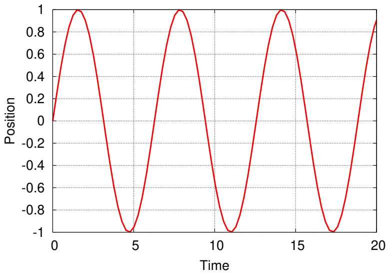


Figure 6.10: The motion of a single metronome arm.

Figure 6.11 shows the motion of four uncoupled metronomes. They move in sine waves with the same frequency, but they're shifted relative to each other because the arms were started at different times.

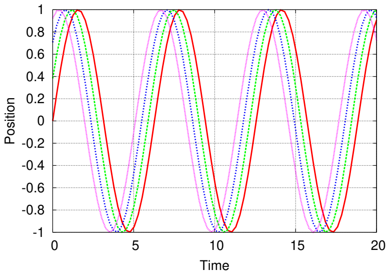


Figure 6.11: The motion of four "uncoupled" metronome arms.

When dealing with oscillating things, it's natural to measure time in terms of multiples of the oscillating period. We could say that the metronome has gone through one cycle, two cycles, three cycles... The vertical axis (arm position) on our graph is the sine of an angle, and the horizontal axis (time) is an angle telling us how far "around" the

[7] http://go.owu.edu/ physics/StudentResearch/2005/BryanDaniels/kuramoto_paper.pdf

cycle we've gone so far. (Note that it's perfecly OK to go around twice, or three times, or as many times as we want.) One complete cycle is equivalent to an angle of 2π radians.

The time to go through one complete cycle is the metronome's period. After some amount of time, t, the "angle" the metronome has traveled through in its cycle is $\theta = 2\pi t / period$. Note that this is different from the physical angle the metronome's arm makes. θ here is an abstract thing that just tells us what stage we're at in the metronome's cycle. If different metronomes are started at different times, that's just equivalent to shifting θ by some amount that we'll call each metronome's "phase angle". When the metronomes jiggle each other, they gradually change each other's phase angles until all they're all the same.

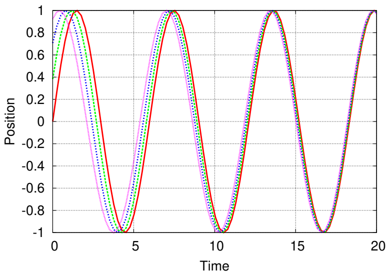


Figure 6.12: The motion of four metronomes that are "coupled" because they're sitting together on a wobbly table.

For coupled metronomes, the Kuromoto model tells us that each metronome is jiggled by each other metronome by an anount that's proportional to the difference in their phase angles. Mathematically, we could say that the change in phase angle of metronome i is:

$$correction_i = \frac{constant}{N} \times \sum_{j=0}^{N-1} (phase_j - phase_i)$$

where N is the number of metronomes and j is a label for each metronome, starting with zero. *constant* is a "coupling strength" that says how strongly the metronomes influence each other[8]. Over time, after many such small corrections, this would cause the phases of the metronomes to converge, as in Figure 6.12.

Program 6.6 tracks the motion of four metronomes that can jiggle each other. It initially gives the metronomes different phase angles spread evenly between zero and $\pi/2$ radians (1/4 of the way through a cycle). Then the program starts a loop that goes through four complete metronome cycles in 100 steps. During each step, the program loops through all the metronomes, and for each metronome it calculates the correction due to all the other metronomes. It then does a second loop and applies those corrections by modifying each metronome's phase angle. During each step, the program prints out the current value of θ and the position of each metronome arm (given by `sin(theta + phase[i])`.

At the top of the program we define two arrays, `phase` and `correction`, that will hold the current phase angle of each metronome and the correction to be applied to that phase angle before beginning the next step. We can't just change the numbers in `phase` because we still need those values until we're finished calculating the correction to each of

[8] In the program below we arbitrarily set this to 0.03. For real metronomes on a real wobbly table, the value of this constant will depend on the wobbliness of the table and other physical parameters.

the metronomes. That's why we store the corrections in a separate array until we're ready to apply them.

When you run the program it will print five columns of numbers: θ, which represents time, and the position of each of the four metronome arms. If you want to simulate more metronomes, just change the value of `nmetronome` in the program.

Program 6.6: metronome.cpp

```
#include <stdio.h>
#include <math.h>
int main () {
  const int nmetronome = 4;
  double nsteps = 100;
  double phase[nmetronome];
  double correction[nmetronome];
  double coupling_strength = 0.03;
  double thetamax = 8.0*M_PI;
  double diff,diffsum;
  double theta,thetastep;
  int istep;
  int i,j;

  diff = 0.5*M_PI/nmetronome;
  for ( i=0; i<nmetronome; i++ ) {
    phase[i] = diff*i;
  }

  thetastep = thetamax/nsteps;
  theta = 0;
   for ( istep=0; istep<nsteps; istep++ ) {

    printf( "%lf ", theta );

    for ( i=0; i<nmetronome; i++ ) {
      printf( "%lf ", sin(theta + phase[i]) );

      diffsum = 0;
      for ( j=0; j<nmetronome; j++ ) {
        diffsum += phase[j] - phase[i];
      }
      correction[i] = coupling_strength*diffsum/nmetronome;
    }

    printf ("\n");

    for ( i=0; i<nmetronome; i++ ) {
      phase[i] = phase[i] + correction[i];
    }

    theta += thetastep;
   }

}
```

Set initial values.

Loop through time

Print θ.

Loop through all metronomes.

Print arm position.

Add up the phase differences.

Apply corrections.

6.10. Multi-Dimensional Arrays

Each array we've seen so far can be visualized as a long, one-dimensional chain of elements, one after another. Arrays don't have to be one-dimensional, though. For example, the program below shows an array called `matrix` with two indices. We could think of this as representing a two-dimensional (20 × 30, in this case) matrix of values.

```
int main(){
  int matrix[20][30];
  int i,j;

  for (i=0; i<20; i++) {
    for (j=0; j<30; j++) {
      matrix[i][j] = i * j;
    }
  }
}
```

The Karl G. Jansky Very Large Array (VLA) is an array of radio telescopes near Socorro, New Mexico. The antennas can turn to follow celestial targets as the Earth rotates. Their motion is usually so slow as to be almost imperceptible, but they periodically need to "unwind" to avoid tangling cables. Astronomers describe the eerie scene when, in the middle of the night, a plain full of antennas suddenly begins twisting in unison, as though they've come to life.

Source: Wikimedia Commons

A two-dimensional array might store the barometric pressure at locations on a map grid, or the number of grains of wheat on each square of a chess board.

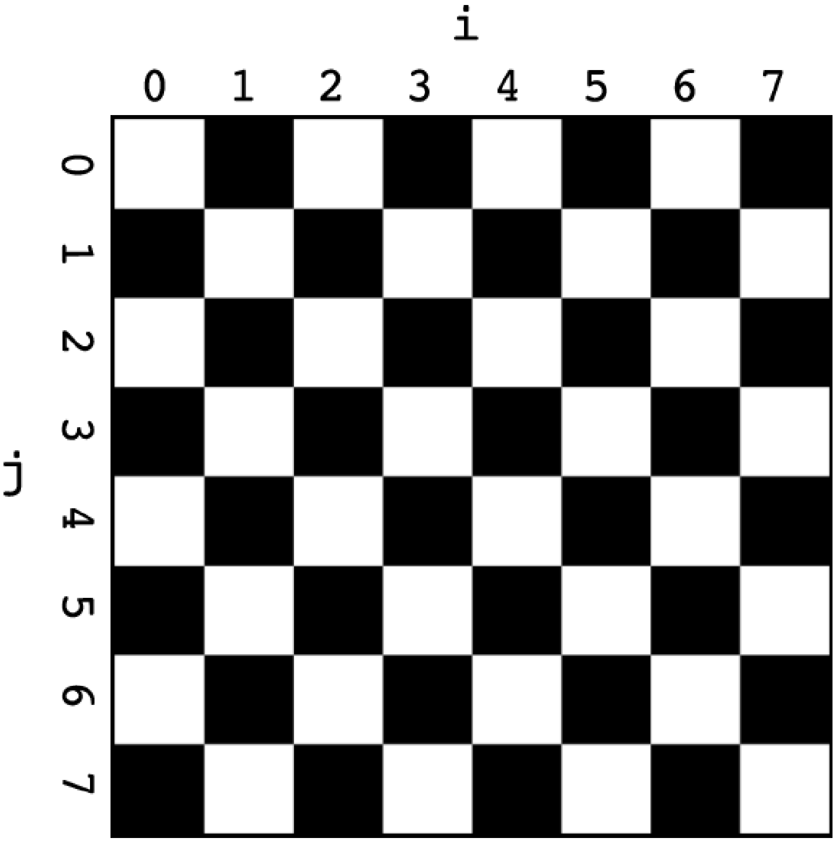


Figure 6.13: An 8 × 8 two-dimensional array, with the indices `i` and `j`.

The Chess Game (1555), by Sofonisba Anguissola

Source: Wikimedia Commons

There's a legend, of uncertain origin, that goes something like this: The inventor of chess presented the new game to his ruler, who was so pleased that he offered the inventor any prize he wanted. The apparently modest inventor asked only for some grains of wheat (or rice, in some versions). One grain was to be placed on the first square of the chessboard, two on the second, four on the third, and so forth, doubling the number of grains each time, until the last square was reached. "Certainly!" said the ruler, but he found that he couldn't honor his offer. To reach the last square would require over 2^{63} grains of wheat, more than all of the wheat in the world!

Arrays in C can have as many indices as you like. A three-dimensional array might hold data about a grid of points in space, or a four-dimensional array might be useful for problems in General Relativity, where space and time are combined into a four-dimensional continuum.

Each index of a multi-dimensional array starts with zero, just like arrays with a single index. In the example above, the first index of `matrix` goes from zero to 19, and the second index goes from zero to 29.

6.11. Working with 2-dimensional Arrays

You're an artillery sergeant in the Union Army during the American Civil War. The rebel forces are trying to float a barge full of coal down the Missisippi river to supply fuel for their new ironclad warship. Your job is to make sure that barge doesn't reach its destination. You set up camp on the side of the river and wait for the barge to come through. But wait! Suddenly a thick fog descends, blocking your view of the river! You'll have to fire blind, and listen for the sound of crackling wood to tell you whether you've hit the barge.

Quickly you sketch out a diagram of the river to help you keep track of hits and misses (see Figure 6.14).

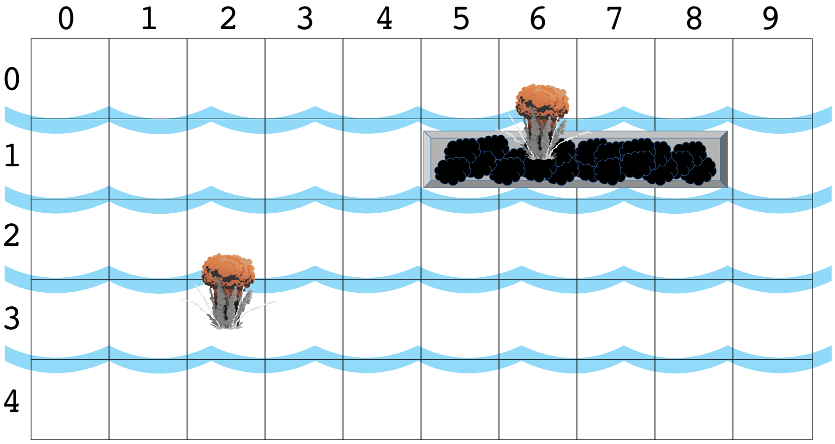


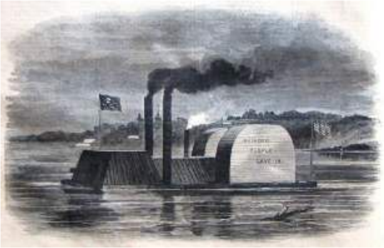
Figure 6.14: A coal barge floats down the Mississippi. The front of the barge is at `[5][1]`, and the vessel occupies the four elements to the right of that position. An artillery shell has hit the barge at position `[6][1]`, but another shell at `[2][3]` has missed.

Hmmm. This sounds like it would make an exciting game! Fortunately, you learned C programming in Boot Camp, so after completing your mission you can return home and write Program 6.7.

These are the rules of the game: A coal barge occupies a line of four consecutive elements in a 2-dimensional array (the map). The barge is oriented horizontally, along the flow of the river, and placed at some random location on the map. The barge must be completely on the map, it can't hang off the edge.

In order to win the game, the player must hit each of the four array elements that contain the barge. The player fires an artillery shell by giving the two indices, `[i][j]`, of an array element. The program tells the player whether the shell hits the barge.

The program uses a 2-dimensional array named `grid` to store a map of the river and the barge's position. Most of this array contains zeros, but the four elements occupied by the barge are initially marked with ones. When a player hits one of the barge elements its value is changed to -1. The variable `nhits` keeps track of the total number of hits. The program keeps running as long as `nhits` is less than four.

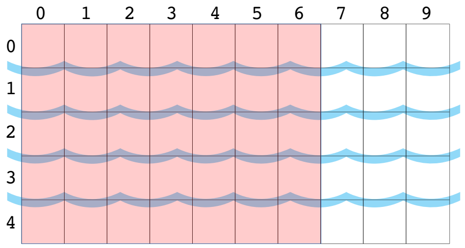


In 1863 Union forces built this dummy ironclad out of an old coal barge, and used it to frighten Confederates. The smokestacks were made of pork barrels and contained smudge pots to make smoke.

Source: Wikimedia Commons

Program 6.7: coalbarge.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
  int grid[10][5];
  int nhits = 0;
  int i, j, iprow, jprow;

  for ( i=0; i<10; i++ ) {
    for ( j=0; j<5; j++ ) {
      grid[i][j] = 0;
    }
  }

  srand( time(NULL) );
  iprow = (int)( 7.0*rand()/(1.0 + RAND_MAX) );
  jprow = (int)( 5.0*rand()/(1.0 + RAND_MAX) );

  for ( i=0; i<4; i++ ) {
      grid[iprow+i][jprow] = 1;
  }

  do {
    printf ("Enter x coordinate: ");
    scanf( "%d", &i );
    printf ("Enter y coordinate: ");
    scanf( "%d", &j );
    if ( i >= 10 || i < 0 || j >= 5 || j < 0 ) {
      printf ("Bad coordinates. Try again.\n");
      continue;
    }

    if ( grid[i][j] == 1 ) {
      printf ("Hit!\n");
      grid[i][j] = -1;
      nhits++;
    } else if ( grid[i][j] == -1 ) {
      printf ("Already hit!  Try again.\n");
    } else {
      printf ("Miss!  Try again.\n");
    }
  } while ( nhits < 4 );

  printf ("You sunk my coal barge!\n");
}
```



The front ("prow") of the barge must be in the shaded section to prevent the back end from hanging off the map.

6.12. Solving a Heat Problem

Let's use a two-dimensional matrix to help us solve a problem. Imagine that you have a square metal plate. One edge of the plate is connected to something very hot, like the engine of a locomotive. The other three edges are connected to a cooling system that keeps them cold. But what are the temperatures of the other parts of the plate?

A steam locomotive.

Source: *Wikimedia Commons*

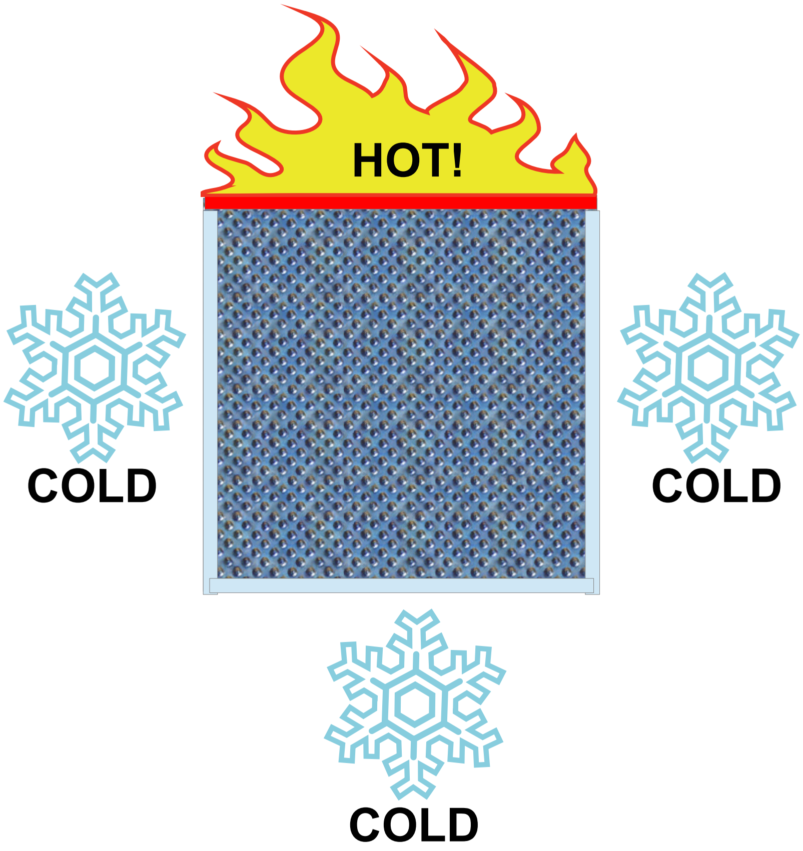


Figure 6.15: A metal plate, hot on one edge and cold on the others.

We might assume that points near each other on the plate would have similar temperatures. Points near the hot edge would tend to be hotter than points near the cold edges. In fact, it wouldn't be surprising if the temperature at any given point were close to the average of the temperatures at the points around it.

Let's write a program that tries to estimate the temperature at various points on the plate. Assume that the very hot edge of the plate has a temperature of 500 celsius, and that the cold sides are kept at a chilly zero celsius by our highly efficient cooling system.

We'll start by dividing the plate into a 100 × 100 grid of points.

Hmmm. . . Sounds like a 2-dimensional array might be useful here. We could define the array like this:

```
double temp[100][100];
```

Pierre-Simon, marquis de Laplace made important contributions to many areas of mathematics.

Source: Wikimedia Commons

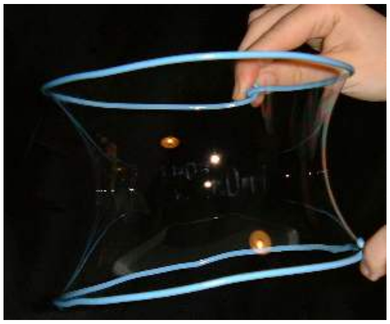
The array named `temp` will hold the temperature values of the points on our grid. We already know the temperatures of some of these points. The points along the hot edge of our plate have a temperature of 500 celsius, and those along the cold edges are at 0. These are the "boundary conditions" of our problem. We need to determine the temperatures of the other, interior, points though.

We'll start by just setting these interior temperatures arbitrarily to zero. This probably isn't a good guess for their temperature, especially for those points near the hot edge, but we can improve our estimate by using the approximation we mentioned above: We're going to assume that the temperature at any point is approximately the average of the temperatures of the neighboring points.

It turns out that this type of problem is a common one in physics and engineering. To arrive at the solution mathematically, we'd need to solve what's called "Laplace's Equation" for this system.[9] Fortunately, there's an easy way to find an approximate solution to Laplace's equation with a computer program. The technique is called "relaxation". You'll see why soon.

After setting the temperatures, let's go through all of the interior points, changing each point's temperature to the average of its neighbors' temperatures. After doing this, we might expect that the temperatures are a now a little more realistic. How far off was our original estimate? We might look at how much difference there is between our original guess and the new estimate. What's the biggest difference?

If we did this averaging process again, we'd get an even better approximation for the temperatures, and we'd see that the maximum temperature change is smaller than it was the first time. If we keep averaging, again and again, the temperature values will eventually settle into stable values that don't change much each time we average. At some point, we decide that this approximation is good enough, and stop averaging.



A soap film stretching between two hoops.

Source: Wikimedia Commons

Our approximation started out very far from the true temperatures.

[9] In the language of math, Laplace's equation is expressed as $\nabla^2\phi = 0$.

You could think of this original approximation as being a rubber sheet that's stretched out into some unnatural shape. Each time we do the averaging process, the sheet "relaxes" a little, until it falls into whatever natural shape fits the boundary conditions we've imposed. That's why this technique is called "relaxation". It can be used for temperature problems like this, but also for a real rubber sheet, or for a soap film on a wire frame. All of these are instances of a system controlled by Laplace's equation.

Program 6.8 follows the strategy we've described above and uses it to find approximate temperatures for interior points on our metal plate. First, it sets temperatures to some initial values, then repeatedly loops through all of the interior points, averaging temperatures. Every time it changes a temperature, it looks to at the size of the change and keeps track of the biggest change. When the changes get small enough (smaller than smalldiff), the program prints the final temperatures.

Notice that Program 6.8 uses some magic to make sure each element of the temp and told arrays contains a value of zero when the program starts. That's what the special value {{0}} means when defining a 2-dimensional array.

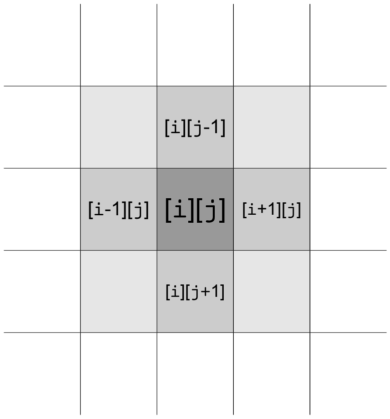


Figure 6.16: Program 6.8 sets the new value of temp[i][j] equal to the average temperature of the four array elements surrounding it. As we saw in Section 6.7, the array indices can be used to tell us something about the relationships between array elements. In this case, the indices give us information about which elements are near each other.

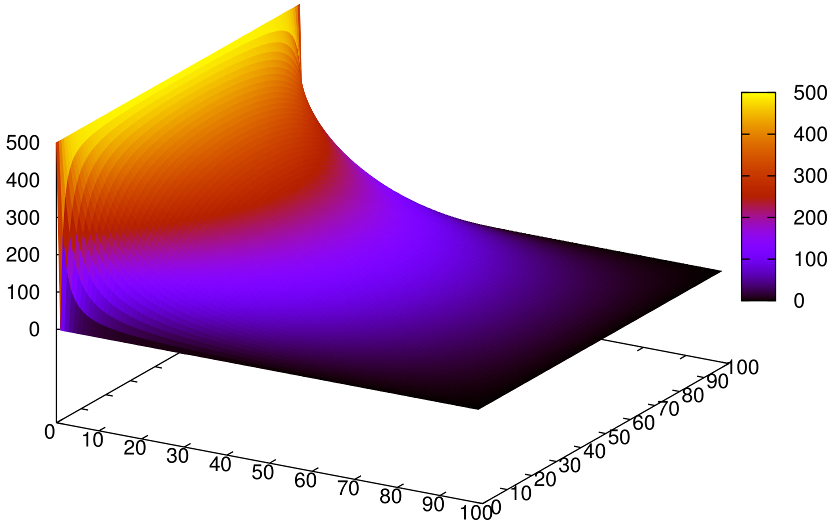


Figure 6.17: Temperatures at various points on the metal plate, as estimated by Program 6.8.

We can put the program's output into a file by writing ./relax > relax.dat, then we can graph the results with *gnuplot*. Figure 6.17 shows the result of the following *gnuplot* command:

```
splot "relax.dat" with pm3d
```

The command `splot`[10]does a "surface plot". The qualifier "`with pm3d`" tells *gnuplot* to use a style of plotting called "palette-mapped 3-d". This color-codes values based on their height. The color scale shows which color corresponds to which value.

To enable *gnuplot* to read the data file, Program 6.8 wrote it in a particular format. If you look inside the data file (`relax.dat`) you'll see that it contains a single column of numbers. If you scroll down in the file a little, you'll see that there's an empty line after every 100 numbers. The numbers represent the temperature values along a row of our grid points on the metal plate. Each extra blank line indicates the beginning of the next row.

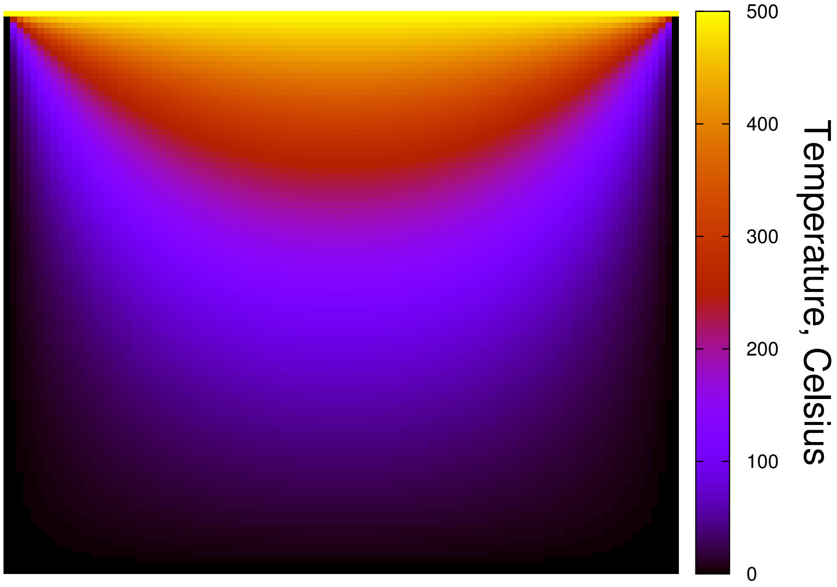


Figure 6.18: Another view of the temperature distribution, as seen from above the plate.

By choosing different "boundary conditions" (the unchanging temperatures at the plate's edges) we can simulate other interesting situations. For example, Figure 6.19 shows the temperature distribution on the plate when there are two hot spots at the top and one hot spot at the bottom.

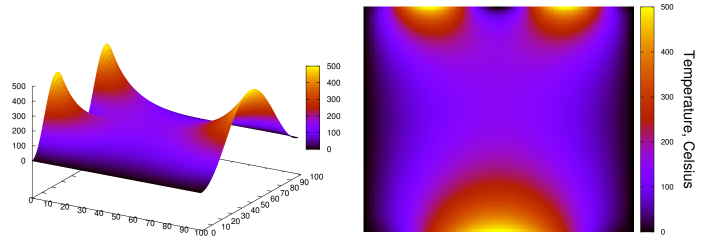
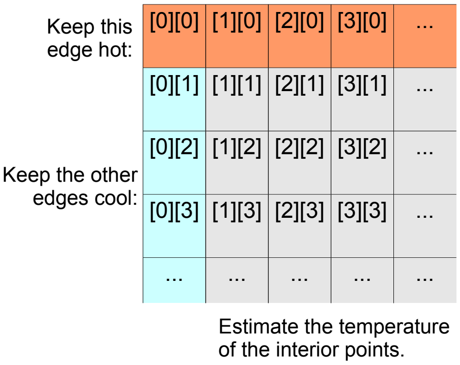


Figure 6.19: These graphs show the temperature distribution when there are two hot spots at the top of the plate and one hot spot at the bottom.

[10] Note that it's **s***plot*, not *plot*.


Keep this edge hot:

Keep the other edges cool:

[0][0]	[1][0]	[2][0]	[3][0]	...
[0][1]	[1][1]	[2][1]	[3][1]	...
[0][2]	[1][2]	[2][2]	[3][2]	...
[0][3]	[1][3]	[2][3]	[3][3]	...
...	...	...	...	...

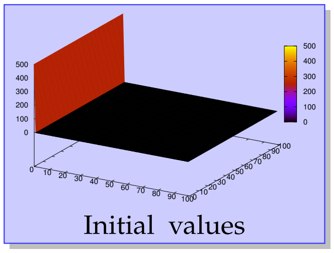
Estimate the temperature of the interior points.

Program 6.8: relax.cpp

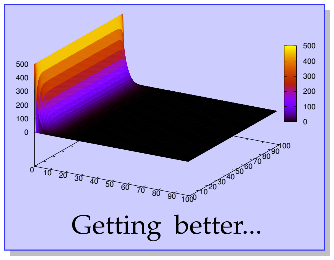
```
#include <stdio.h>
#include <math.h>
int main () {
  int i,j;
  double diff, maxdiff, smalldiff=1.0e-3;
  double temp[100][100] = {{0}}; // Current temps.
  double told[100][100] = {{0}}; // Previous temps.

  // Set elements along hot edge to 500 celsius:
  for (i=0;i<100;i++){
    temp[i][0] = 500.0;
  }

  // Keep averaging until maxdiff is small:
   do {
    for (i=0;i<100;i++){
      for (j=0;j<100;j++){
        told[i][j] = temp[i][j];
      }
    }

    maxdiff = 0;
    // These two nested loops go through all of the
    // interior points:
    for (i=1;i<99;i++){
      for (j=1;j<99;j++){
        temp[i][j] = 0.25 * (told[i-1][j] + told[i+1][j] +
                             told[i][j-1] + told[i][j+1]);
        diff = fabs(temp[i][j]-told[i][j]);
        if ( diff > maxdiff) {
          maxdiff = diff;
        }
      }
    }
   } while ( maxdiff > smalldiff );

  // Write out results:
  for (i=0;i<100;i++){
    for (j=0;j<100;j++){
      printf("%lf\n", temp[i][j]);
    }
    printf ("\n");
  }
}
```

Relax...

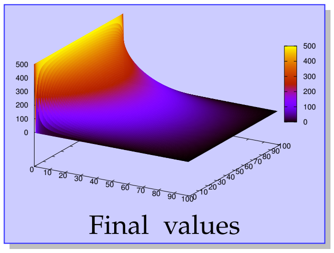
See Figure 6.16 on Page 203



Initial values



Getting better...



Final values

But what about. . . ?

Let's look more closely at the trick we used in Program 6.8 when defining the `temp` and `told` arrays. If a statement like:

```
double temp[100][100] = {{0}};
```

gives each of the array's elements a value of zero, could we do this:

```
double temp[100][100] = {{100}};
```

to set all of the elements to 100?

Unfortunately, no, but the real result might suprise you. If you printed the values stored in an array defined like this, you'd find that element `[0][0]` had the value 100, as expected, but all of the other elements would be set to zero.

Let's back up a little and see how these curly brackets work when we use them in an array definition. As we noted back on Page 181, we can initialize the elements of an array by explicitly giving a list of values in curly brackets, like this:

```
int marbles[5] = {7,9,3,15,8};
```

But what if the list contains fewer values than the number of array elements, like this?:

```
int marbles[5] = {7,9};
```

In that case, the first two elements would be set to 7 and 9, and the rest would be set to zero. Whenever there are too few values, the computer assumes that we want to set the rest of the values to zero.

As we've noted before (see Chapter 4), variables are just named sections of the computer's memory, and we can't assume that they contain any particular value before we explicitly give the variable a value. If we want all of an array's elements to be zero, we need to set them to zero. We could do this by saying:

```
int marbles[5] = {0,0,0,0,0};
```

or, as we've just seen, we could say:

```
int marbles[5] = {0};
```

causing the computer to set the first element to zero, and setting all of the other elements to zero by default, since we didn't say what we wanted them to be.

Some compilers will even let us say "`int marbles[5] = {}`", but that doesn't work with all of them, so it's best to always give at least one value.

So what about the double brackets we used in Program 6.8? That's because these are 2-dimensional arrays. With a 2-d array, we can initialize values like this:

```
double x[20][20] = {{7,9},{4,3}};
```

setting the first two elements of the first row to 7 and 9, and the first two elements of the second row to 4 and 3. All of the other elements would be set to zero. And, if we said:

```
double x[20][20] = {{0}};
```

all of the array's elements would be set to zero. This is the trick we used in Program 6.8.

Figure 6.20: A collection of marbles within the permanent collection of The Children's Museum of Indianapolis.

Source: Wikimedia Commons

6.13. Conclusion

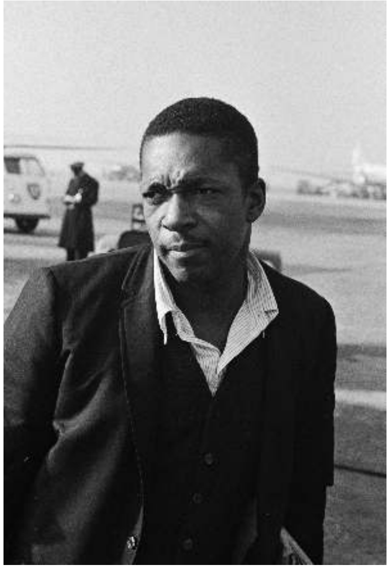
Arrays are useful whenever your program needs to store several related values. Array indices uniquely identify each array element, and they may also say something about relationships between array elements. (They can indicate the spatial or time ordering of measurements, for example.)

Some important things to remember about arrays are:

- The elements of an array can be of any type (but all elements of a given array must be of the same type).
- When defining an array, the number in square brackets says how many elements are in the array.
- It's important to remember that an N-element array's indices start with zero, and end at N-1.
- Arrays take up memory. It's easy to write "double x[1000]", but remember that this takes as much memory as a thousand single variables. Keep this in mind when defining large arrays.
- Array elements can be referred to by their indices.
- The index must be an integer.
- The index uniquely identifies a single array element.
- Arrays can optionally be initialized when they're defined.

You load sixteen tons and what do you get? Another day older and deeper in debt.
11
12

Practice Problems



John Coltrane. Because "Coal Train".

Source: Wikimedia Commons

1. On page 185 it was suggested that adding an "`if`" statement to Program 6.2 could make it safer. Add an "`if/else`" statement to Program 6.2 (without changing anything else!) to check whether the number is too big or too small. If it is, ignore the number and give the user a helpful message. Call your program `coal.cpp`.

2. Create, compile and run Program 6.8. Use the "`ls`" command to make sure that the program created the file `relax.dat`.

   Use the *gnuplot* command described above to plot the data using *gnuplot*'s "pm3d" plotting style. If your version of *gnuplot* allows it, grab the figure with your mouse and rotate it around. Does it look like what you'd expect?

3. Imagine you have a very short coal train, containing only five cars. Each of the cars is to be sent to one of your customers. Each customer is identified by an integer "Valued Customer ID Number" (VCID) like "37654".

   (a) Using *nano*, create a data file named `orders.dat` that contains five rows of numbers, one row for each car in your train. Each row of the file will have two numbers: the weight of coal in that car (which might be a number with decimal places), and the ID of the customer it belongs to (which will always be an integer). The file might look like this:

   ```
   63.4 5487
   52.1 30978
   77.8 8765
   89.0 435
   95.3 789
   ```

   (b) Now write a program named `orders.cpp` that will read `orders.dat`. All of the weights should go into a 5-element array of `double`s named `carweight` and all of the customer IDs should go into a 5-element array of `int`s named `vcid`, so that `carweight[0]` and `vcid[0]` are the weight and customer ID for the first car, and so forth. After the program reads the data, have it ask the user for a car number (a number between zero and four) and print out the weight and customer ID for that car. Make sure you **check the car number** to see if it's in the range zero to four, and tell the user if it's not. Also make sure the program tells the user which number is which.

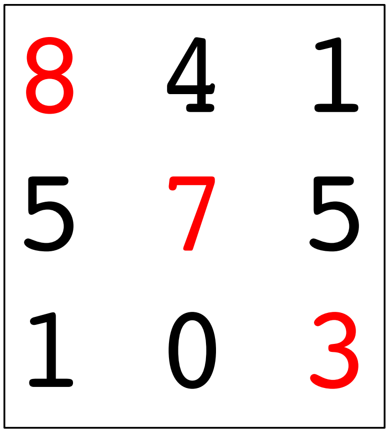
   Hints: See Program 6.4 for something similar, and look at Chapter 5 for advice about reading things from files.

Problem 3 uses two "parallel arrays", `carweight` and `vcid`, to store two pieces of information about each car. If we wanted to add more information (maybe the car's age, so we know when to replace it?) we could add more arrays. We'll see a different way to do this sort of thing later, in Chapter 12.

4. In mathematics, a *matrix* is an array of numbers. Matrices are important in many fields of science, engineering and mathematics.

   Using *nano*, create a file named `matrix.dat` that contains a 3×3 matrix like this:

   ```
   8 4 1
   5 7 5
   1 0 3
   ```



8 4 1
5 7 5
1 0 3

Figure 6.21: The *trace* of a matrix is defined as the sum of its diagonal elements. In the example above, the trace is equal to $8+7+3$.

   Write a program named `matrix.cpp` that reads data from `matrix.dat` and puts the numbers into 2-dimensional array, `double m[3][3]`. To do this, use nested pair of `for` loops that repeatedly uses `fscanf` to read the array's elements, one at a time. The `fscanf` statement might look like this:

   ```
   fscanf( input, "%lf", &m[col][row] );
   ```

   where `col` and `row` are the column and row numbers.

   Make the program do the following:

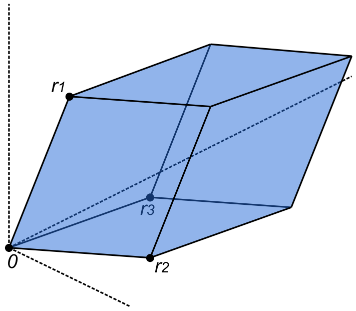
   (a) First, print out the elements of the matrix, so you can make sure they match the data in `matrix.dat`. The numbers should be printed in rows and columns, like Figure 6.21.

   (b) Second, compute and print out the *trace* of the matrix. The trace is defined as the sum of the matrix's diagonal elements. (See Figure 6.21.) (**Hint:** The diagonal elements are the ones where the row and column numbers are the same, like `m[0][0]` and `m[1][1]`.)

   (c) Third, compute and print out the *determinant* of the matrix. The determinant for a 3×3 matrix is:

   ```
   det =
     m[0][0]*( m[1][1]*m[2][2] - m[1][2]*m[2][1] ) +
     m[0][1]*( m[1][2]*m[2][0] - m[1][0]*m[2][2] ) +
     m[0][2]*( m[1][0]*m[2][1] - m[1][1]*m[2][0] );
   ```

   You'll obviously need to be careful when typing this into your program, but looking at the way the numbers in the statement line up vertically can help you avoid mistakes.



We can think of each row of the matrix as the coordinates of a point in three-dimensional space. In the picture above, we call these points *r*1, *r*2, and *r*3. The *determinant* of the matrix is just the volume of the parallelepiped defined by these three points (the shaded volume above).

Source: Wikimedia Commons

   If you make your `matrix.dat` file look like the example above, you should find that the matrix has a trace of 18 and a determinant of 121. Use these values to check your work.

5. Write a program named `fibarray.cpp` that fills an array with the first 20 terms of the Fibonacci sequence. The first two numbers in the Fibonacci sequence are 1, 1, and each subsequent number is the sum of the preceding two numbers. Your program should have a 20-element `int` array named `term`. The program should start out by setting `term[0]=1` and `term[1]=1`. Then the program should have a "`for`" loop that finds the value of each of the remaining 18 terms. Inside the loop you'll want to have a statement like `term[i+2] = term[i]+term[i+1]`. After this loop is done, add another loop that prints out all the elements of `term`. The program's output should look like this:

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

6. In the Fibonacci sequence each term is the sum of the two preceding terms. There's also a "Tribonacci sequence", in which each term is the sum of the *three* preceding terms. It starts out with the numbers 0, 0, 1. Referring to the instructions in Problem 5, write a program named `tribarray.cpp`, but with the Tribonacci numbers instead of the Fibonacci numbers. The program's output should look like:

```
0 0 1 1 2 4 7 13 24 44 81 149 274 504 927 1705 3136 5768 10609 19513
```

7. A black-and-white image can be represented by a two-dimensional array of ones and zeros. Each element of the array represents a pixel of the image, and the array indices (say, `i` and `j`) are the x and y coordinates of that pixel. Wherever the image is white, the array element contains a zero, and where it's black there's a one. If we assume that all the pixels are white unless we say otherwise, we could store the image as a list of the `i` and `j` indices of all the black pixels, like this:

```
0 4
0 5
1 4
1 5
3 1
```

and so forth.

A vision in black and white.

Source: Wikimedia Commons

First, fetch the data file named `banner.dat` by typing either:

```
wget http://tinyurl.com/practical-c/data/banner.dat
```

or

```
curl -L -O http://tinyurl.com/practical-c/data/banner.dat
```

whichever works on your computer. This file contains the `i` and `j` indices of all the black pixels in an 80×24-pixel image. In this file `i` represents the horizontal position of each pixel (measured left-to-right) and `j` represents the vertical position (measured top to bottom).

Then write a program named `image.cpp` that draws the image by printing an asterisk for each black pixel in the image and a space for each white pixel. The program should define a 2-d array named `pixel` like this:

```
int pixel[80][24] = {{0}};
```

(This uses the trick we used in Program 6.8 (`relax.cpp`) to initially set all of the elements to zero.) Then use a loop to read pairs of `i` and `j` coordinates from `banner.dat` and set `pixel[i][j] = 1`. After reading all of the data from `banner.dat` use a pair of nested loops to go through all of the elements of `pixel`, printing a `*` or space as appropriate. Remember to print a newline when you get to the end of each row. When you run your program, you should see a picture like this:

```
********  **     ** **    **  ******  ****  ******   ******
**     ** **     **  **  **  **    **  **  **    ** **    **
**     ** **     **   ****   **        **  **       **
********  *********    **     ******   **  **        ******
**        **     **    **          **  **  **             **
**        **     **    **    **    **  **  **    ** **    **
**        **     **    **     ******  ****  ******   ******
```

8. Imagine that you're a game designer and you want to rotate a picture of a box. (See Figure 6.22.) You have the coordinates of the box's corners. What math magic can you apply to this data in order to rotate the box by a given angle around the origin? It turns out that matrix multiplication is very useful for this.

   The trick is to think of each corner as a vector (a pair of x and y coordinates), and then multiply that vector by something called a "rotation matrix" to create a new vector giving the point's position after rotation. If we call the original vector $\mathbf{v}$, new vector $\mathbf{w}$, and the rotation matrix $\mathbf{m}$ we can say this mathematically as:

$$\mathbf{w} = \mathbf{m} \cdot \mathbf{v}$$

   The components of $\mathbf{m}$ look like this:

$$\begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$$

   where θ is the angle we want to rotate by. In a computer program we could represent this matrix with a 2-dimensional 2×2 array:

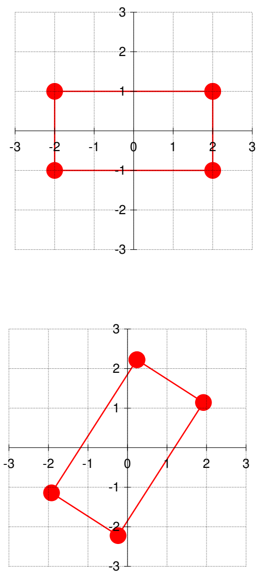


Figure 6.22: The top figure shows a rectangle that we want to rotate. The bottom figure shows it after the rotation.

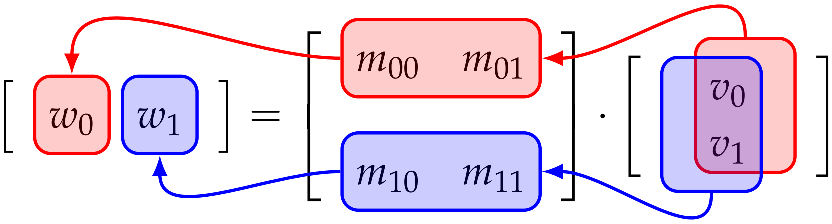
```
double m[2][2];

m[0][0] =  cos(theta);
m[0][1] = -sin(theta);
m[1][0] =  sin(theta);
m[1][1] =  cos(theta);
```

and we could represent each of the vectors as a 1-dimensional array with 2 elements, like:

```
double v[2];
double w[2];
```

where `v[0]` is the x component of **v** and `v[1]` is the y component.

So how do we multiply a vector by a matrix? Take a look at the figure below:



$$w_0 = m_{00} \cdot v_0 + m_{01} \cdot v_1$$

$$w_1 = m_{10} \cdot v_0 + m_{11} \cdot v_1$$

When we multiply a vector times a matrix, the matrix tells us how to mix together the components of the original vector to make a new vector.

Write a program named `rotate.cpp` that asks the user to enter a rotation angle, in radians, then opens a file named `rectangle.dat` and reads positions of a rectangle's corners from the file. The content of `rectangle.dat` is shown in Figure 6.23. Each time the program reads a point, use the rotation matrix to calculate the new position of the point after it has been rotated and print the new x and y coordinates, separated by a space.

The file `rectangle.dat` should contain this data:

```
2 1
-2 1
-2 -1
2 -1
2 1
```

Figure 6.23: The coordinates of the corners of the rectangle shown at the top of Figure 6.22. Note that the last point is just a copy of the first point, so that we can draw the rectangle by giving *gnuplot* the commands:

```
set size ratio -1
plot "rectangle.dat"with lines lw 10
```

(Think about what would happen if we didn't go back to the first point.)

9. Write a program named `vecadd.cpp` that adds vectors together and tells you their sum. The program should let users enter as many vectors as they want. To be adventurous, let's assume that these are 4-dimensional vectors, but let's write our program so that it's easy to change the number of dimensions later. For example, we might define our arrays like this:

```
const int dimension = 4;
double vector[dimension];
double sum[dimension] = {0};
```

Notice that we've used the `{0}` trick described on page 206 to set all the elements of `sum` to zero.

Your program should use a "`do-while`" loop. Every time it goes around the loop it should ask the user to enter the components of a vector, then it should ask if the user wants to enter another vector (with 1=yes, 0=no). The "`while`" statement at the end of the loop should check to see if the user entered a 1 or a 0. Each time the user enters a new vector it should be added to the current sum of all the vectors.

Four dimensions are OK, but here's The 5th Dimension looking extra groovy in 1969.

Source: Wikimedia Commons

For letting the user enter the elements of a vector, we can do something like this:

```
for ( i=0; i<dimension; i++ ) {
  printf( "%d: ", i );
  scanf( "%lf", &vector[i] );
  sum[i] += vector[i];
}
```

Notice how this adds to `sum` each time a new component is entered.

At the end of the program, it should use another "`for`" loop to print the components of the sum of all the vectors in an easy-to-read format.

10. Write a program named `avgscore.cpp` that reads a gradebook and tells you the average score for a given assignment. Start by using `nano` to create a data file named `scores.dat` that contains five rows of ten scores, like this:

```
94.0 87.0 85.5 92.5 82.5 82.5 99.5 84.0 89.0 85.5
84.0 97.0 97.5 94.0 87.5 80.0 95.5 98.0 96.5 93.0
91.0 98.0 85.5 96.5 91.0 88.5 99.5 90.5 93.0 89.0
96.5 97.0 86.0 94.0 91.5 94.0 94.0 87.5 92.0 91.0
80.5 96.5 82.0 93.0 92.5 88.5 97.0 93.5 98.5 81.0
```

(You can just copy these scores if you like.) Each row represents one student's work on each of ten different assignments. Your program should ask users which assignment they're interested in[11], then it should read the data file and compute the average score for that assignment. At the end it should print the average score on the screen in a nice way (for example, "The average score for assignment 5 is 87.85.").

The program should read one row at a time, using a ten-element

[11] Be sure to check that it's a number between 0 and 9, and give an error message if it isn't.

array to temporarily store that row's scores. Assume that there will always be exactly five students and exactly ten assignments.

Hint: When reading a row, you might do something like this:

```
for ( assignment=0; assignment<10; assignment++ ) {
    fscanf( input, "%lf", &scores[assignment] );
}
```

11. Let's write a simple game program named `sortit.cpp`. This program will generate a list of ten random numbers, and ask the player to sort them in ascending order by swapping pairs of numbers. During each turn, players will be asked which two numbers they want to swap, and then the program will show the player how the list looks after swapping the numbers.

A street in the town of Sort, in Catalonia, Spain.
Source: *Wikimedia Commons*

The program should start by filling a ten-element array with random integers. Then it should give the user some instructions and show the list of numbers, along with their array indices. The program should print something like this:

```
Your job is to sort the numbers from small to big.
 0: 2084671678
 1: 839235044
 2: 591510472
 3: 1963580438
 4: 1014188406
 5: 29385297
 6: 730613533
 7: 969172124
 8: 1969905057
 9: 635261567
which two numbers do you want to swap? (-1 to quit)
```

At this point, the program should use two `scanf` statements[12] to read the index numbers of the two elements. If the user enters -1 as the first element number, the program should stop.

If the user enters two index numbers, the program should swap those two elements of the array, then show what the modified array looks like and offer the user the opportunity to swap two more elements. See the program `bubble.cpp` on Page 192 for an example of how to swap two elements of an array. The program should continue offering the user the opportunity to swap elements until the user decides to quit, by entering -1 as the first element number.

[12] Can you see why we need two separate `scanf` statements instead of one `scanf` that reads both numbers at once?

After you've written this program and played with it a little bit, you might think about how this game could be improved. How would you modify it to keep score, for example? How would you modify it to tell the user when the array is properly sorted?

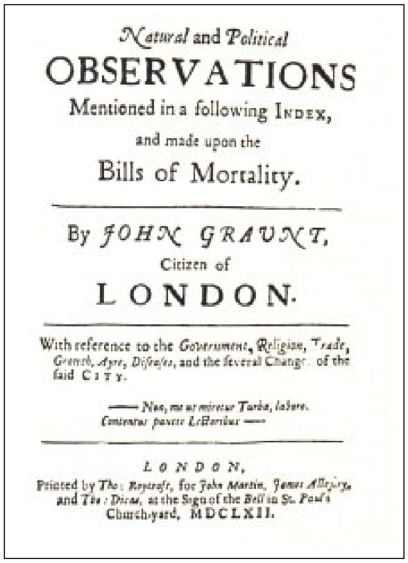
7. *Statistics*

Figure 7.1: *Der Sommer*, by Abel Grimmer (565-1630).

Source: Wikimedia Commons

7.1. Introduction

In the 17th century, English authors John Graunt and William Petty began writing about a new science called "Political Arithmetic", which tried to understand social, economic, and public health problems through the collection and analysis of numerical data. In the 18th century, authors such as Germany's Gottfried Achenwall began writing about another new field of study called "Statistik" which aimed at discovering the general principles by which a state could be successfully run.



Natural and *Political*
OBSERVATIONS
Mentioned in a following INDEX,
and made upon the
Bills of Mortality.

By *JOHN GRAUNT*,
Citizen of
LONDON.

With reference to the *Government*, *Religion*, *Trade*, *Growth*, *Ayre*, *Diſeaſes*, and the ſeveral Changes of the ſaid CITY.

—— Non, me ut miretur Turba, laboro,
Contentus paucis Lectoribus ——

LONDON,
Printed by *Tho: Roycroft*, for *John Martin*, *James Allestry*, and *Tho: Dicas*, at the Sign of the *Bell* in *St. Paul's* Church-yard, MDCLXII.

Figure 7.2: John Graunt's *Observations on the Bills of Mortality* (1662) studied mortality data in an effort to understand the spread of Bubonic Plague.

Source: Wikimedia Commons

Statistik soon began using the techniques of Political Arithmetic. The success of a state might depend on the amount of wheat or milk it produces, or the number of skilled craftsman. A spreading plague might be detected by systematically collecting data about deaths. These studies were the beginning of what we call "statistics" today.

The modern science of statistics attempts to see inaccessible underlying truths by sampling the superficial things that are visible to us. By surveying a limited number of households, we arrive at an estimate of the total number of families living in poverty. By observing a few thousand particle decays, we estimate the probability that such decays will happen. In the language of Antoine de St. Exupery's *Little Prince*, statistics tries to see the elephant that lies hidden inside the boa (see Figure 7.3).

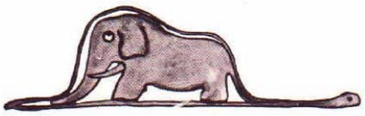


Figure 7.3: A boa who's swallowed an elephant, from Antoine de St. Exupery's *The Little Prince*.

The available data is often incomplete, and shows us only a blurry outline of what's underneath, so statistics also tries to measure the uncertainty in its estimates. These measures of uncertainty help us judge how much we should trust our statistical conclusions.

7.2. Summarizing Data with Histograms

It can be hard to see the patterns in a bunch of raw numbers, but a graph often makes the data snap into focus. In this section, we'll look at a new kind of graph called a "histogram". The histogram was introduced in 1891 by Karl Pearson, one of the founders of modern statistics. It summarizes an arbitrarily large amount of data by reducing it to a smaller, fixed, number of data points that represent how often certain values appear in the original data.

Figure 7.4: British mathematician Karl Pearson (1857-1936).

Source: Wikimedia Commons

Let's look at an example. Particle physicists often use "scintillation detectors" to measure the energy of subatomic particles. A "scintillator" is a material such as Thallium-doped Sodium Iodide which produces a flash of light when an energetic particle passes through it. By measuring this flash of light, we can find out how much energy is deposited as a particle passes through. More light means more energy.

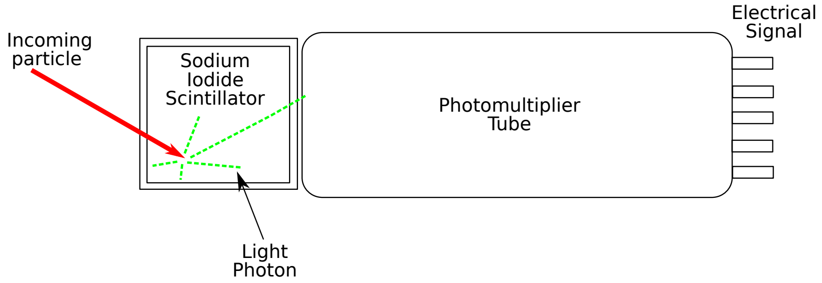


Figure 7.5: A scintillation detector produces a flash of light whenever an energetic particle passes through it. The amount of light is proportional to the energy that the particle deposits in the detector. The flash of light is converted into an electrical signal by a "photomultiplier tube", and the electrical signal is measured and recorded.

The output of such a detector is just a bunch of numbers, each of which corresponds to the energy deposited by a detected particle.[1] These energies are measured in "electron Volts" (eV), and a million electron volts is called an MeV. The data we collect might look like Figure 7.6.

It's hard to see patterns in a stream of numbers like this, but let's imagine that we've looked at the data and noticed that all of the numbers lie between 0 and 20 MeV. It would be interesting to know how the numbers are distributed in this range. Are they spread uniformly? Do they bunch up in some places?

```
15.130490
16.942571
16.627112
10.780935
14.569799
15.192141
 6.489004
12.386759
17.793823
 4.181682
19.381618
...
```

Figure 7.6: Some data from our detector, representing energies measured in MeV. It's hard to make sense of a stream of numbers.

If we were rather bad at programming but good with tools, we might construct a set of bins like those in Figure 7.7 to satisfy our curiosity. Each bin represents a 4 Mev-wide range of energies. Whenever we see a particle with an energy in that range, we could drop a marble into the corresponding bin. After going through all of the data we could look at our bins and easily see which energies were the most common, because they'd contain the most marbles.

[1] The size of the electrical signals coming out of the detector is proportional to the energy. For our example, we'll just assume that we can read the energy values directly.

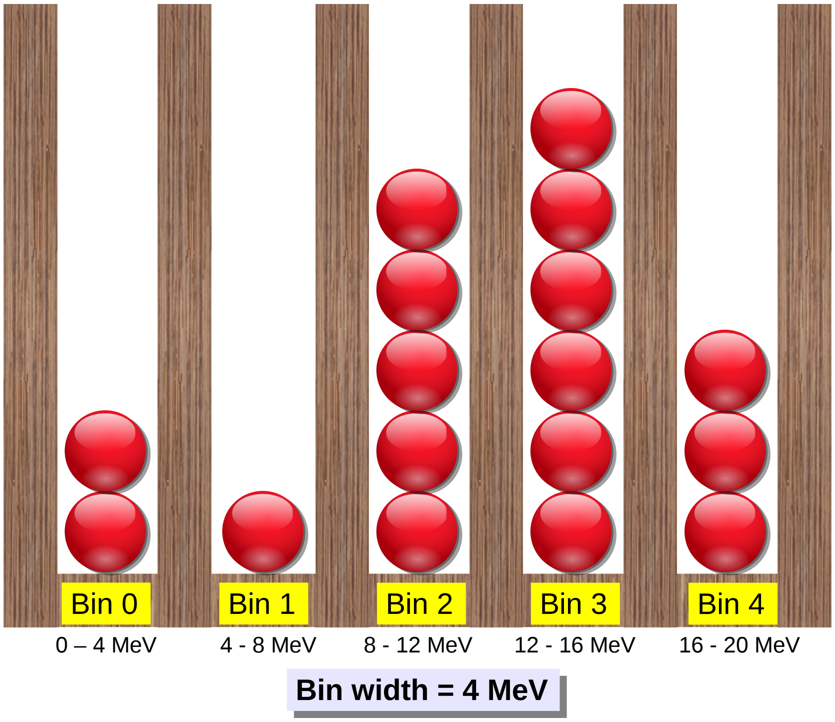


Figure 7.7: Binning the detector data produces a histogram.

The pattern of high and low marble stacks that we've produced is called a histogram. It tells us how frequently a measurement falls within a given range. For this reason, histograms are sometimes called "frequency plots".

If we wanted to save our histogram (maybe we want to re-use the lumber for another project?) we could just write down the number of marbles in each bin. But if a histogram is just equivalent to a list of numbers, that means we could use an array in a C program to store it.

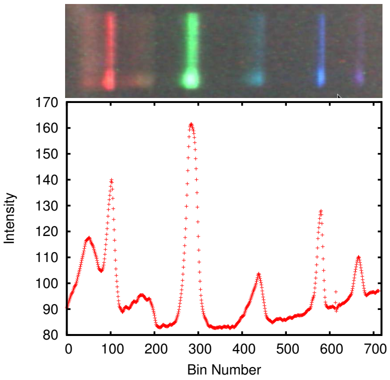


Figure 7.8: A histogram can also represent a spectrum. The most intense places on this fluorescent light spectrum are just those where photons are most frequent. In the graph, we've marked only the top of each of 700 "columns of marbles".

Spectrum taken by Finian Wright, using a DIY spectrometer.

Program 7.1 reads energies from a file and produces a histogram, represented by an array of bin counts. The program reads a list of numbers from the file `energy.dat`. The numbers represent energies from a scintillation counter, ranging between approximately 0 and 50 MeV. For each number, the program adds a virtual marble to one of 50 bins. The bins are the elements of the array named `bin`.

To find out which bin to put the marble into, the program divides each energy value by the bin width, and rounds the result down to the nearest integer. The result is the bin number. For example, take a look at Figure 7.7 again. In this figure, an energy of 9 MeV would go into bin number 2, since the bin width is 4 MeV, and $9/4 = 2.25$.

In Program 7.1, for simplicity, we've made the bin width 1 MeV, so we can just look at the bin number to see the approximate energy it represents.

Program 7.1: hist.cpp

```
#include <stdio.h>
int main () {
  int i, binno, overunderflow = 0;
  double energy, binwidth = 1.0;
  int bin[50];
  FILE *input;

  for ( i=0; i<50; i++ ) {
    bin[i] = 0; // Reset all bins to zero.
  }

  input = fopen( "energy.dat", "r" );
  while ( fscanf( input, "%lf", &energy ) == 1 ) {
    binno = energy/binwidth; // Find which bin.

    // Is it too small or too big?
    if ( binno < 0 || binno >= 50 ) {
      overunderflow++;
      continue; // Skip this value and jump to the next.
    }

    bin[binno]++; // Add a marble to this bin.
  }
  fclose(input);

  for ( i=0; i<50; i++ ) {
    printf ("%d %d\n", i, bin[i]);
  }
  printf ("# Saw %d over/underflows\n", overunderflow);
}
```

Read lines from file.

At the end of the program, it prints out each bin number and the number of virtual marbles that bin contains.

As we saw in Chapter 6, it's important to check our array indices to make sure we're not going past the end of the array. What if the file `energy.dat` contains some unexpected energies that would fall into bins beyond the last element of our `bin` array? What if a negative number somehow found its way into the file? We'd want to know about these things, but we wouldn't want our program to crash.

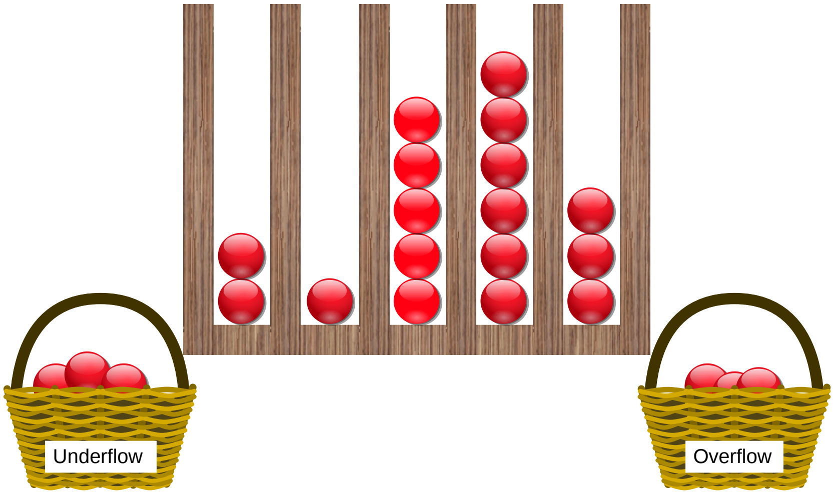


Figure 7.9: In Program 7.1, `overunderflow` counts the number of overflows and underflows.

To record these unexpected values, Program 7.1 has a variable called `overunderflow` that counts the number of overflows (energies that are too low) and overflows (energies that are too high). The program checks the energy with an "`if`" statement like this:

```
if ( binno < 0 || binno >= 50 )
```

The condition in the "`if`" statement checks to see if either of two conditions are true by using the "`or`" operator, `||`. (We say `>= 50` because the highest bin number is 49.)

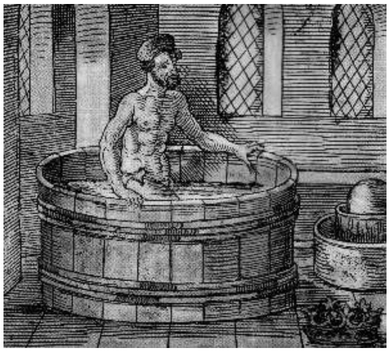


Figure 7.10: Legend has it that the Greek philosopher Archimedes proved the value of noticing overflows. He'd been given the task of measuring the density of a crown to determine whether it was made of pure gold. This required measuring the crown's volume, but he couldn't figure out how to do that. Getting into his bath one day, he noticed that his body displaced an equal volume of water, and it was easy to measure the volume of water. He jumped from the tub, shouted "*Eureka!*", meaning "I've found it!" and ran naked through the streets of Syracuse.

Source: Wikimedia Commons

If an overflow or underflow is found, the program increments the value of `overunderflow` and then immediately skips to the next energy value in `energy.dat`. It accomplishes this by using a "`continue`" statement. In Chapter 4 we saw that it was possible to stop a loop by using a `break` statement. The `continue` statement is similar, except that instead of stopping the loop, it causes the program to skip the rest of the current trip through the loop and immediately start the next trip.

When the program finishes, it prints out the number of overflows and underflows that were seen. Notice that it prints a hash symbol (#) in front of the message about over/underflows. This is so the message won't confuse *gnuplot* if we want to plot the results. *Gnuplot* ignores any lines that begin with #.

Exercise 36: Making a Histogram

For this exercise you'll need a copy of the data file `energy.dat`. You can find instructions for obtaining it in Appendix C.2 on page 622. Take a look inside this file using *nano*. You should see a single column of numbers, representing simulated energy measurements of 100,000 particles.

```
35.130490
36.942571
36.627112
40.780935
34.569799
35.192141
36.489004
32.386759
...
```

Figure 7.11: Some of the data in the file `energy.dat`.

Try graphing this file by starting *gnuplot* and typing:

```
plot "energy.dat"
```

The result should look something like Figure 7.12.

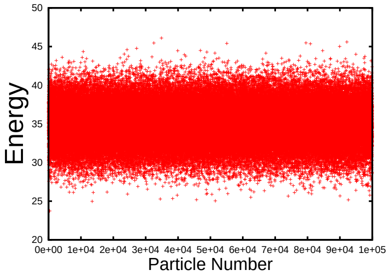


Figure 7.12: The data in `energy.dat`, plotted with *gnuplot*.

Exit from *gnuplot* and then create, compile and run Program 7.1. The program's output should be two columns of numbers (a bin number and the number of "virtual marbles" in that bin), followed by a message about overflows and underflows. By looking at the columns of numbers, you should already be able to see a pattern emerging.

Now run the program again, redirecting its output into a file, like this:

```
./hist > hist.dat
```

Start *gnuplot* and plot the data by using the command:

```
plot "hist.dat" with impulses
```

"`with impulses`" causes *gnuplot* to draw a vertical line for each point. The result should look something like Figure 7.13. Where do most of the energy values lie?

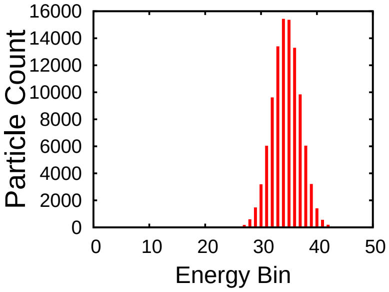


Figure 7.13: The output of Program 7.1, plotted with *gnuplot*.

Even though the data file we're analyzing (`energy.dat`) contains 100,000 lines, the output of Program 7.1 is just two 50-line columns. We could give Program 7.1 a million times more data to analyze, and the program's output would still be only fifty lines, although the numbers on those lines would be larger. This is one reason histograms are useful: they can summarize large data sets very efficiently. In the exercise above, the program turns 100,000 numbers into a 50-number summary.

7.3. Resolution and Range of Histograms

We could improve Program 7.1 by making a few changes that allow us to adjust the *resolution* of the histogram (the width of its bins) and its *range* (the lowest and highest energy values it can display). Let's also make the program more general, so it's clear we can use it for other kinds of data besides energy values.

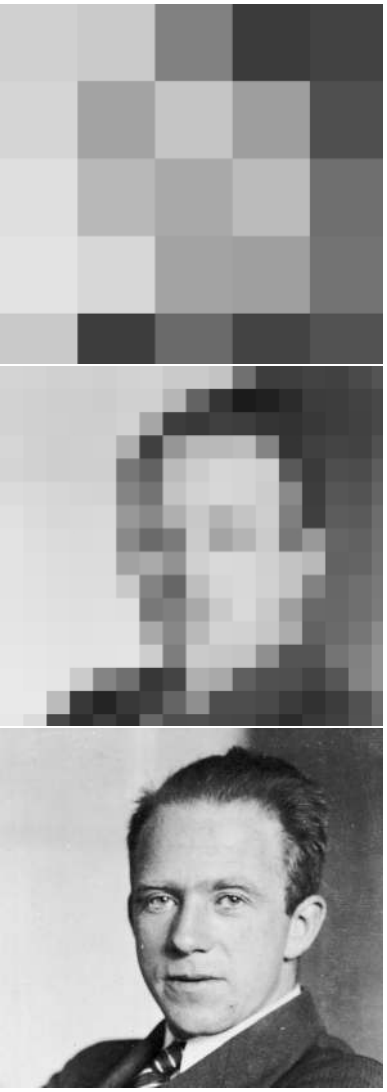


Figure 7.14: Finer-grained resolution sometimes shows us features of our data that are invisible at lower resolutions. (Photo of Werner Heisenberg.)

Source: Wikimedia Commons

Controlling the Resolution of a Histogram:

In Program 7.1 we set the bin width to 1 MeV for convenience, so we could see the energy values by just looking at the bin number. Bin number 35 corresponded to 35 MeV. What if we wanted a finer- or coarser-grained histogram, though? We might want a bin width of 0.5 MeV or 2 MeV, for example. In that case, we might want the program to print the *energy value* of each bin instead of the bin number.

But do we want to print the energy at the left side of the bin, the right side, or the middle? These are all different. Let's just print all of them, and then we can decide which value we want to use when we graph the data.

We can make this happen by modifying just a few lines of our program. Instead of saying this:

```
printf ("%d %d\n", i, bin[i]);
```

we can say this:

```
elow = binwidth*i;
emid = binwidth*(0.5+i);
ehi = binwidth*(i+1);
printf ("%lf %lf %lf %d\n", elow, emid, ehi, bin[i]);
```

The first three lines calculate the energy value at the left, center, and right of the bin (to get the center, we add 0.5 to the bin number). Then, instead of printing the bin number, we print all three energy values. This will mean that our output has four columns: the three energy values and the number of "marbles" in the corresponding histogram bin.

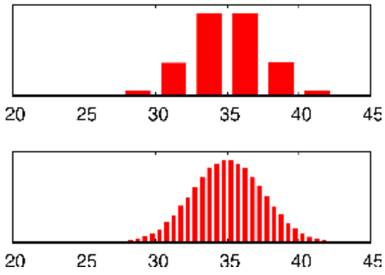


Figure 7.15: A low-resolution histogram (top) with 10 bins, and a high-resolution histogram (bottom) with 50 bins, both showing the same set of data.

Controlling the Range of a Histogram:

Program 7.1 also assumes that the energy range we're interested in starts at zero. Sometimes this won't be the case. Maybe we want to focus on the range between 30 and 40 MeV, for example. Or, if we're measuring something other than energy, we might even have negative values. Maybe we're measuring distance, and we want to look at values between -10 meters and 10 meters, where zero is the origin of our coordinate system.

Figure 7.16: Two images with the same resolution (both are 348×348), but the bottom image zooms in on a small region near the center of the upper image. If we have a fixed number of histogram bins, we should try not to waste them on regions where there's no interesting data. (Image of a "gnat ogre" – a robber fly of the genus *Holcocephala* – taken by the author.)

To accommodate that we'll need to make a few more changes to our program. First, let's define the lower bound of our energy range with a new variable:

```
double emin = 20.0; //MeV.
```

Here we've set it to 20 MeV, but we could set it to whatever we want. Now we'll need to use this value when we calculate the bin number (binno) and when we calculate the energy of each bin at the end of the program. Our new calculation of binno would look like this:

```
binno = (energy-emin)/binwidth;
```

Instead of just energy, we're using energy-emin to determine which bin we should use. When energy is equal to emin, the bin number is zero. At the end of the program, when calculating the left, center, and right energy values of the bin we can say:

```
elow = emin + binwidth*i;
emid = emin + binwidth*(0.5+i);
ehi = emin + binwidth*(i+1);
```

We've added emin because the lowest bins correspond to that energy.

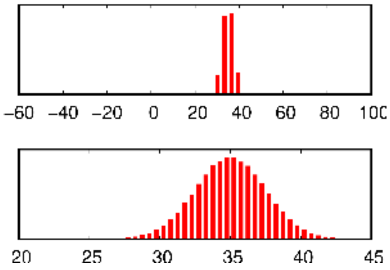


Figure 7.17: A histogram with a large range (top), and a small range (bottom). Both show the same set of data using the same number of bins.

Calculating `binwidth` *Instead of Specifying It:*

It's often convenient to specify the limits of a histogram's range and the number of bins, and then let the program calculate the value of binwidth. We might, for example, want 100 bins covering the range from 20 MeV to 45 MeV. That would tell us that each bin has a width of $(45 - 20)/100 = 0.25$ MeV.

We'll need to rearrange a few things to make that happen. Let's start by adding a new variable to specify the upper end of our range:

```
double emax = 45.0; //MeV.
```

Now let's define a variable that specifies the number of bins, to make it easy to adjust this value later:

```
const int nbins = 50;
int bin[nbins];
```

As we mentioned in Chapter 6, the word `const` tells the C compiler that this value should never change. (See Page 186.) Next, we need to add a line to calculate the value of `binwidth`:

```
binwidth = (emax-emin)/nbins;
```

Finally, we need to replace `50` with `nbins` wherever the program has previously assumed there were 50 bins.

Putting It All Together:

Okay, now let's see what the finished program looks like after we've made all of these changes. Notice that Program 7.2 uses `x`, `xmin` and `xmax` in place of `energy`, `emin` and `emax`, since we can use this program for any kind of data. There are also some new `printf` statements at the bottom of the program that remind the user about the program's settings.

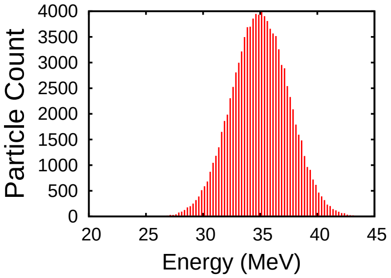


Figure 7.18: Output of Program 7.2, plotted with the *gnuplot* comand `plot "hist.dat" using 2:4 with impulses`.

Program 7.2: hist.cpp, Improved

```
#include <stdio.h>
int main () {
  int i, binno, overunderflow = 0;
  double x, xlow, xmid, xhi, binwidth;
  double xmin = 20.0;
  double xmax = 45.0;
  const int nbins = 100;
  int bin[nbins];
  FILE *input;

  binwidth = (xmax-xmin)/nbins;

  for ( i=0; i<nbins; i++ ) {
    bin[i] = 0; // Reset all bins to zero.
  }

  input = fopen( "energy.dat", "r" );
  while ( fscanf( input, "%lf", &x ) == 1 ) {
    binno = (x-xmin)/binwidth;
    if ( binno < 0 || binno >= nbins ) {
      overunderflow++;
      continue; // Skip this value and jump to the next.
    }
    bin[binno]++; // Increment the appropriate bin.
  }
  fclose(input);

  for ( i=0; i<nbins; i++ ) {
    xlow = xmin + binwidth*i;
    xmid = xmin + binwidth*(0.5+i);
    xhi = xmin + binwidth*(i+1);
    printf ("%lf %lf %lf %d\n", xlow, xmid, xhi, bin[i]);
  }
  printf ("# Xmin = %lf\n", xmin);
  printf ("# Xmax = %lf\n", xmax);
  printf ("# Binwidth = %lf\n", binwidth);
  printf ("# Nbins = %d\n", nbins);
  printf ("# Saw %d over/underflows\n", overunderflow);
}
```

7.4. Two-Dimensional Histograms

Imagine that you're a school principal whose students have just finished taking reading and math tests. You could make a histogram of all the reading scores or all the math scores, but you'd like to see how reading scores and math scores are related to each other. Do students with high math scores also have high reading scores, or do students excel in only one area? What can we do? Let's stroll down the hall and talk to the Shop teacher. He's a clever guy. Maybe he'll have a suggestion.

You begin by telling him about the wooden bin you constructed for sorting marbles in the preceding section. He thinks about the problem for a moment, then says, "Well, all you need to do is make a crate that lets you sort marbles out in two directions: one direction for reading scores and the other for math. Give me a few minutes and I'll make one for you." Sure enough, after a few minutes of sawing and hammering, he's produced a crate like the one shown in Figure 7.19.

"Great!" you say. "Each marble represents a student. I just need to drop the marble into the bin that corresponds to that student's reading and math scores. In the end, the number of marbles in a bin will tell me how many students had that particular combination of reading and math scores."

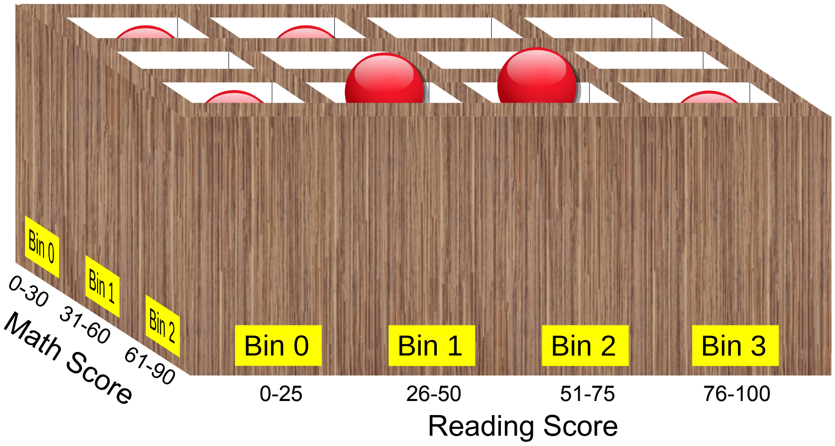


Figure 7.19: Binning marbles in two directions produces a two-dimensional histogram. In this example, math scores range from zero to 90 and reading scores range from zero to 100. We've divided the math scores into bins with a width of 30, and the reading scores into bins with a width of 25.

Our crate full of marbles can be thought of as a two-dimensional histogram[2]. As with the one-dimensional version we saw in the preceding section, we can save our histogram by just writing down the number of marbles in each bin. In Program 7.1 we used a one-dimensional array (`bin[50]`) to hold the values in our one-dimensional energy histogram. For a two-dimensional histogram, we'll need a *two-dimensional* array. We might store the number of marbles in each bin of Figure 7.19 in a 3×4 array of integers, defined like this: `int bin[3][4];`

[2] Two-dimensional histograms are sometimes called "bivariate" histograms, because they show data from two variables (reading score and math score in this example).

Take a look again at Program 7.1 (hist.cpp). If we wanted to modify this program so that it makes a two-dimensional histogram, we'd need to change bin into a 2-d array, and we'd need to modify the way we fill this array.

For example, assume we have a data file that has two numbers on each line: a math score and a reading score. Instead of the single bin number (binno) that we calculate in Program 7.1, we now need to calculate two bin numbers, one for math and one for reading. We might do that like this:

```
mbin = math/mbinwidth;
rbin = reading/rbinwidth;

if ( rbin < 0 || rbin >= nrbins ||
     mbin < 0 || mbin >= nmbins ) {
  overunderflow++;
  continue; // Skip this value and jump to the next.
}

bin[mbin][rbin]++; // Increment the appropriate bin.
```

where reading and math are the reading and math scores, mbin and rbin are the calculated bin numbers for math and reading, mbinwidth and rbinwidth are the widths of the math and reading bins, and nmbins and nrbins are the number of math and reading bins.

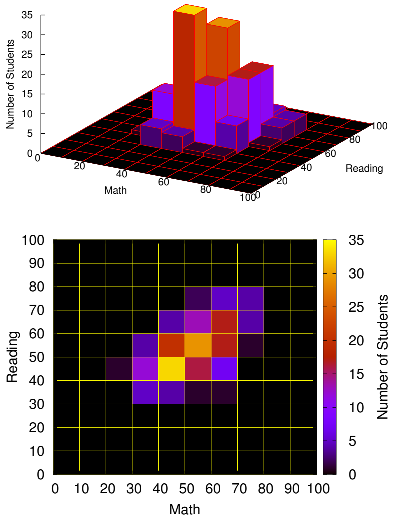


Figure 7.20: Two ways we might represent the data in a two-dimensional histogram.

Figure 7.20 shows two ways of representing a 2-dimensional histogram of reading and math scores. Here the reading and math scores both range between zero and 100, and we've split each range into ten bins. In the top picture, we use a vertical bar to represent the height of each bin's stack of marbles. In the bottom picture we look down on the top of these stacks, and we've color-coded each stack to indicate its height.

Two-dimensional histograms are useful when we want to see how two measured quantities interact with each other. In Figure 7.20, we can easily see that students with high math scores also tend to have high reading scores. This wouldn't be obvious if we just looked at the numbers, or graphed math or reading scores by themselves.

7.5. Finding the Mean

Looking at the one-dimensional histogram in Figure 7.13 we can see that the energies tend to cluster around approximately 35 MeV, but they trail off to the left and right in a bell-shaped curve. If all of the particles actually had the same energy, and all of their energy was deposited in the detector, we might expect all of the numbers in `energy.dat` to be exactly the same. In practice, though, our measurements will always have some random variation no matter how careful we are. This is partly because of imperfections in our instruments, but there may also be physical limits to the precision of our measurements

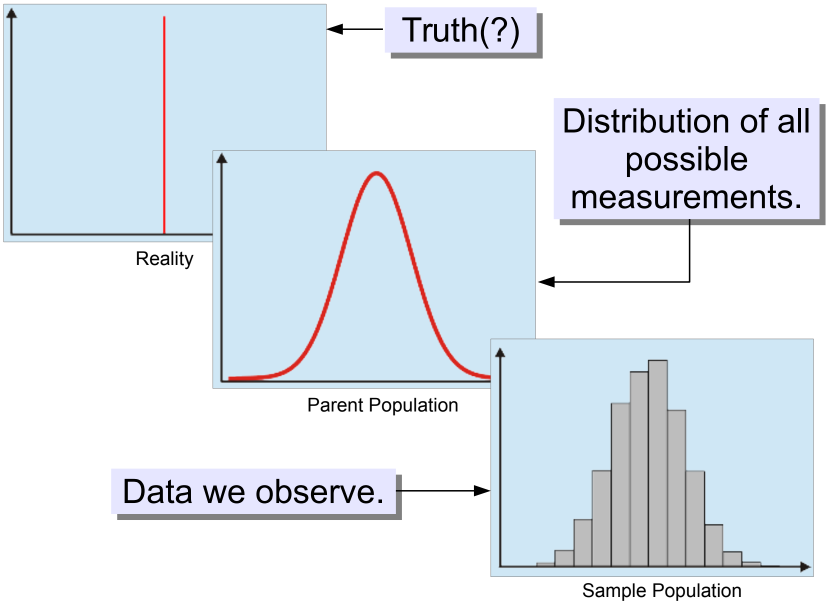


Figure 7.21: We are always at two removes from the "underlying truths" that we're trying to measure. Statisticians call the right-hand graph the "sample population", and the middle graph the "parent population", from which the sample is drawn at random.

If we made an infinite number of measurements, we might see that they're spread out like the middle graph in Figure 7.21. In reality, we make a finite number of measurements that are just a small sample of all of the possible measurements, like the right-hand graph. If we only take a few measurements, it's not too unlikely that all of them may happen to lie on the left or right side of the true value. As we make more measurements, our data will begin to look more and more like the middle graph.[3]

Once we've taken enough measurements to approximate the middle graph, what's our best guess for the true value in the left-hand graph? Some of our measurements are higher than the true value and some are lower, but we expect that the true value lies somewhere between the extremes, at some "average" value.

[3] In statistics, this is called "The Law of Large Numbers".

In everyday speech, we use the word "average" to mean "typical". The "average guy" is a typical person. How do we measure this, though? How can we *objectively* decide what's "typical"?

Figure 7.22: *The Tempting Cake*, by Albert Rosenboom

Source: *Wikimedia Commons*

In science, we often use a quantity called the "arithmetic mean" (often just called the "mean") to represent what's "average" or "typical". You've probably used this before. The mean of a set of values is the sum of all the values, divided by the number of values. Mathematically, we could write it like this:

$$\overline{X} = \frac{1}{N}\sum_{i=1}^{N} X_i \qquad (7.1)$$

where N is the number of values, X_i are the values themselves, and $\overline{X}$ is the mean.

If we slice a cake into several pieces, the *mean size* of a piece is the sum of the size of all the pieces (which is just the total size of the cake), divided by the number of pieces. The *mean* is the size that each piece would have if the cake were sliced up into perfectly equal parts.

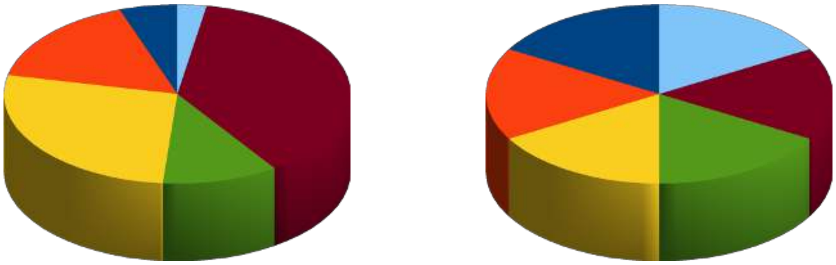


Figure 7.23: On the left, an unfairly sliced cake. On the right, a cake sliced into equal pieces. The size of each right-hand slice is equal to the *mean size* of the left-hand slices.

We often assume that the mean value of our measurements is the best guess at the true, underlying value that we're trying to measure. If we make enough measurements, we expect that the mean value will approximate the mean value of all possible measurements, and we expect that the mean of all possible measurements will approximate the true, underlying value, which may never be directly accessible to us.

Program 7.3 reads the energy values from `energy.dat` and finds their arithmetic mean. In the program, the variable named `sum` is intially set equal to zero. Each time a new number is read, it's added to `sum`. After reading all of the numbers, the program calculates the mean by dividing the sum by the number of energy values.

Program 7.3: mean.cpp

```
#include <stdio.h>
int main () {
  double energy;
  double sum = 0.0;
  int nvalues = 0;
  double mean;
  FILE *input;

  input = fopen("energy.dat","r");
  while ( fscanf( input, "%lf", &energy ) == 1 ) {
    sum += energy;
    nvalues++;
  }

  mean = sum/nvalues;

  printf ("Number of values is: %d\n", nvalues );
  printf ("Mean value is: %lf\n", mean );

  fclose (input);
}
```

We could also modify our histogram program (Program 7.1) so that it tells us the mean energy. Program 7.4 is a new version of `hist.cpp` that adds up the energy values as they're read, and prints out the mean when it's done. Again, we put a # on the front, so *gnuplot* will ignore this line.

Notice that we want to include all of the energy values, even the underflows and overflows. We want the arithmetic mean of *all* values.

Exercise 37: You Big Meanie!

Modify your earlier `hist.cpp` program so that it looks like Program 7.4. Compile and run it. Does the value given by the program look consistent with what you saw when you plotted a histogram of the data (Figure 7.13)?

Figure 7.24: In the 1968 Beatles movie *The Yellow Submarine*, the Blue Meanies *hated* music.

Source: *unigami, at Deviant Art*

Program 7.4: hist.cpp, Version 2

This version of the program prints the average energy value. Changes from Program 7.1 are shown in bold.

```
#include <stdio.h>
int main () {
  int i, binno, overunderflow = 0;
  double x, xlow, xmid, xhi, binwidth;
  double xmin = 0.0;
  double xmax = 50.0;
  double sum = 0.0;
  int nvalues = 0;
  const int nbins = 50;
  int bin[nbins];
  FILE *input;

  binwidth = (xmax-xmin)/nbins;

  for ( i=0; i<nbins; i++ ) {
    bin[i] = 0; // Reset all bins to zero.
  }

  input = fopen( "energy.dat", "r" );
  while ( fscanf( input, "%lf", &x ) == 1 ) {

    sum += x;
    nvalues++;

    binno = (x-xmin)/binwidth;
    if ( binno < 0 || binno >= nbins ) {
      overunderflow++;
      continue; // Skip this value and jump to the next.
    }
    bin[binno]++; // Increment the appropriate bin.
  }
  fclose(input);

  for ( i=0; i<nbins; i++ ) {
    xlow = xmin + binwidth*i;
    xmid = xmin + binwidth*(0.5+i);
    xhi = xmin + binwidth*(i+1);
    printf ("%lf %lf %lf %d\n", xlow, xmid, xhi, bin[i]);
  }
  printf ("# Xmin = %lf\n", xmin);
  printf ("# Xmax = %lf\n", xmax);
  printf ("# Binwidth = %lf\n", binwidth);
  printf ("# Nbins = %d\n", nbins);
  printf ("# Saw %d over/underflows\n", overunderflow);
  printf ("# Mean value is %lf\n", sum/nvalues );
  printf ("# Nvalues = %d\n", nvalues );
}
```

Add each value to the sum.

Count the number of values.

7.6. Standard Deviation

Figure 7.13 shows that the energy values in `energy.dat` tend to bunch up in one spot, forming a peak. If you were describing this shape to someone, you could start by telling them that "the mean energy value is 35 MeV". This says where the peak is, but it doesn't tell them anything about how wide it is. How can we measure the width of a peak like this?

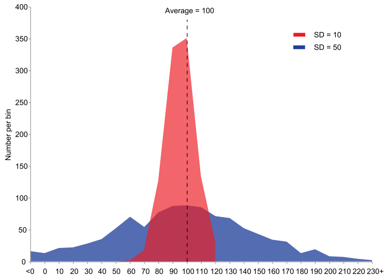


Figure 7.25: A comparison of histograms made from two samples, one with a small standard deviation and one with a large standard deviation. Both samples have the same mean value and contain the same number of data points.

Source: Wikimedia Commons

If the peak is wide, we might expect that a lot of data points would be far from the mean value. In the terms used in Equation 7.1, we might think about going through all of the points and adding up the values of $X_i - \overline{X}$. Unfortunately, we'd find that this sum is always zero, since some points are to the left of the mean and some to the right. It's possible to prove mathematically that the sum of all of these positive and negative distances will always add up to zero.

What we really want is just the distance from the mean, without worrying about whether it's positive or negative. Since the square of a real number is always positive, we might think about adding up the squares of the $X_i - \overline{X}$ values. Statisticians define a quantity called the "sample variance" that does just this. It's defined this way[4]:

$$s^2 = \frac{1}{N-1}\sum_{i=1}^{N}(X_i - \overline{X})^2 \qquad (7.2)$$

where s^2 is the variance. For the example we've been working on, the units of the variance would be MeV^2 (energy squared). The square root of the variance is called the "standard deviation".[5] In our example, this has units of MeV, and it can be used to describe the width of the peak in Figure 7.13. The standard deviation tells us the "typical" distance between a data point and the mean value.

Figure 7.26 shows some data along with its arithmetic mean ($\overline{X}$) and standard deviation (s). The data we observe is just a sample of all the possible values we might see if we did an infinite number of measurements. Our data is called the "sample" and the collection of all possible values is called the "parent". Underneath it all, like the elephant inside the boa, is the true value that we're trying to estimate.

There's a practical problem with using Equation 7.2 in a computer program, though. Since it uses $\overline{X}$ (the mean value of the energy), we'd have to first loop through all of the energy values to calculate their mean, and then loop through them all again to calculate the variance.

[4] Why do we divide by $N-1$ instead of N? A simple explanation is that the variance is undefined if you have only one data point.

[5] This is another term that was introduced in the 1890s by Karl Pearson.

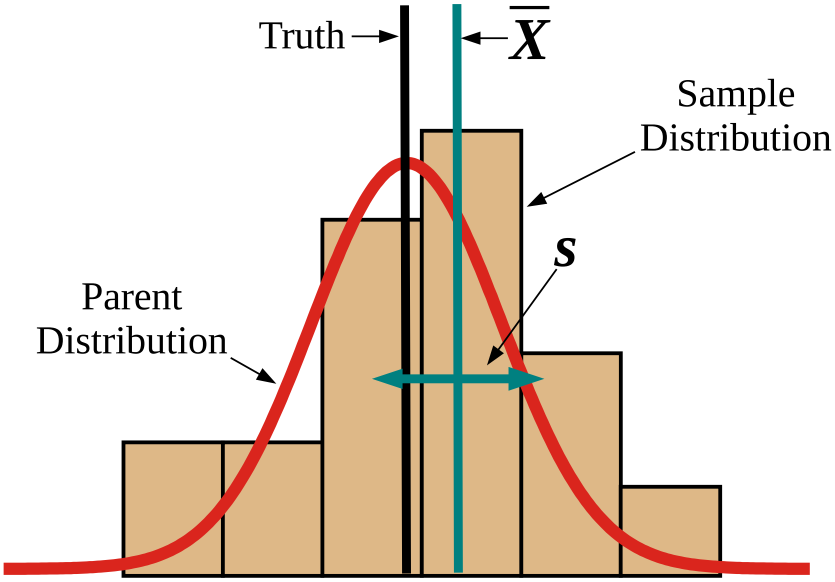


Figure 7.26: Sample distribution, parent distribution (the set of all possible measurements), and true value. $\overline{X}$ is the mean of the sample, and s is its standard deviation.

Fortunately, clever mathematicians have provided us with a shortcut to make things easier. It turns out that Equation 7.2 can be rewritten like this:

$$s^2 = \frac{1}{N-1}\left[\sum_{i=1}^{N} X_i^2 - \frac{1}{N}(\sum_{i=1}^{N} X_i)^2\right] \tag{7.3}$$

The right-hand sum in Equation 7.3 is the same one we're already using in Program 7.4. To find the variance we also need the left-hand sum, which is the sum of the *squares* of the values. Our program just needs to do one loop, and keep two sums: the sum of the values and the sum of their squares.

That's what Program 7.5 does with our `energy.dat` data. The program includes `math.h` at the top, since it uses the `sqrt` and `pow` functions. We've also added a new variable `sum2` to store the sum of the squares, from Equation 7.3. At the end of the program, we calculate the standard deviation and print it out.

Program 7.5: stddev.cpp

```
#include <stdio.h>
#include <math.h>
int main () {
  double energy;
  double mean;
  double stddev;
  double sum = 0.0;
  double sum2 = 0.0;
  int nvalues = 0;
  FILE *input;

  input = fopen("energy.dat","r");
  while ( fscanf( input, "%lf", &energy ) == 1 ) {
    sum += energy;
    sum2 += pow( energy, 2 );
    nvalues++;
  }

  mean = sum/nvalues;
  stddev = sqrt( (sum2 - sum*sum/nvalues)/(nvalues-1) );

  printf ("Number of values is: %d\n", nvalues );
  printf ("Mean value is: %lf\n", mean );
  printf ("Std. Dev is: %lf\n", stddev );

  fclose (input);
}
```

This program is an improved version of `mean.cpp` (Program 7.3) that prints out the standard deviation of the energy values. Changes from Program 7.3 are shown in bold.

We can apply the same technique to our ever-improving `hist.cpp` program, giving it the ability to print out the standard deviation as well as the mean value. That's what we do in Program 7.6.

Exercise 38: Finding the Standard Deviation

Create, compile, and run Program 7.6, a new version of `hist.cpp` that now prints the standard deviation. How large is this value in comparison with the width of the peak in Figure 7.13?

This is an updated version Program 7.4. Changes from Program 7.4 are shown in bold.

Program 7.6: hist.cpp, Version 3

```
#include <stdio.h>
#include <math.h>
int main () {
  int i, binno, overunderflow = 0;
  double x, xlow, xmid, xhi, binwidth;
  double xmin = 0.0;
  double xmax = 50.0;
  double sum = 0.0;
  double sum2 = 0.0;
  int nvalues = 0;
  const int nbins = 50;
  int bin[nbins];
  FILE *input;

  binwidth = (xmax-xmin)/nbins;

  for ( i=0; i<nbins; i++ ) {
    bin[i] = 0; // Reset all bins to zero.
  }

  input = fopen( "energy.dat", "r" );
  while ( fscanf( input, "%lf", &x ) == 1 ) {

    sum += x;
    sum2 += pow( x, 2 );
    nvalues++;

    binno = (x-xmin)/binwidth;
    if ( binno < 0 || binno >= nbins ) {
      overunderflow++;
      continue; // Skip this value and jump to the next.
    }
    bin[binno]++; // Increment the appropriate bin.
  }
  fclose(input);

  for ( i=0; i<nbins; i++ ) {
    xlow = xmin + binwidth*i;
    xmid = xmin + binwidth*(0.5+i);
    xhi = xmin + binwidth*(i+1);
    printf ("%lf %lf %lf %d\n", xlow, xmid, xhi, bin[i]);
  }
  printf ("# Xmin = %lf\n", xmin);
  printf ("# Xmax = %lf\n", xmax);
  printf ("# Binwidth = %lf\n", binwidth);
  printf ("# Nbins = %d\n", nbins);
  printf ("# Saw %d over/underflows\n", overunderflow);
  printf ("# Mean value is %lf\n", sum/nvalues );
  printf ("# Std. dev. is %lf\n",
          sqrt( (sum2 - sum*sum/nvalues)/(nvalues-1) ) );
  printf ("# Nvalues = %d\n", nvalues );
}
```

Needed for sqrt and pow.

Add square of each value to sum2.

7.7. The "Normal" or "Gaussian" Distribution

The peak in Figure 7.13 is a bell-shaped curve. Curves like this occur very frequently in data. In fact, they occur so frequently that this shape is called the "Normal Curve". The German mathematician Carl Friedrich Gauss (1777-1855) was perhaps the first to appreciate the significance of it, so it's sometimes called a "Gaussian Curve".

Figure 7.27: Gauss is pictured on this German banknote. If you look closely you'll see a small picture of the Normal curve at the left.

Source: Wikimedia Commons

The ubiquity of this curve was a source of amazement to early statisticians, who saw it popping up everywhere: astronomical data, actuarial tables, agricultural data.

Why does this curve appear so often? Because of the "Central Limit Theorem", which says that any linear sum of random variables tends toward a Normal distribution, no matter what the distribution of the individual variables looks like.[6]

The Central Limit Theorem is so important that it's called the "second fundamental theorem of probability". (The first is the Law of Large Numbers.)

The Normal curve can be expressed mathematically by the following equation:

$$P(x) = Ae^{-\frac{(x-\overline{x})^2}{2s^2}} \tag{7.4}$$

The curve reaches its maximum at $\overline{x}$, the mean value of x. The curve's width is controlled by s, the standard deviation. The height of the curve at its maximum is A.

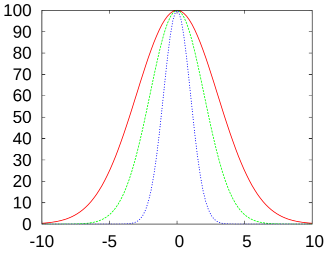


Figure 7.28: Three Normal curves with standard deviations of 3 (the widest), 2 and 1.

If we look at data that's bunched together in a Normal distribution, the standard deviation of the data gives us some quantitative information about the way the data is distributed. We know, for example, that about 68% of Normally-distributed data lies within one standard deviation away from the mean value. (See Figure 7.30.)

If Program 7.6 tells us that the standard deviation of our energy data is 2.5 MeV and the mean is 35 MeV, that implies that 68% of our energy values fall between 32.5 MeV and 37.5 MeV. If we were telling someone about our measurements, we might say that the energy value we observed was 35 ± 2.5 MeV.

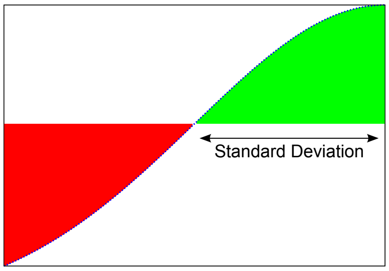


Figure 7.29: The standard deviation of a Normal curve is the horizontal distance from the midline to one of the points where the curvature changes from positive to negative.

[6] Note that this means you can construct a pretty good Normal distribution just by adding together sufficiently many numbers pulled from any random distribution. For example, roll six dice and add their numbers together. Keep doing this and recording the sum each time. A histogram of the sums will look very similar to the Normal distribution.

We also know that about 95% of the data lie within 2 standard deviations from the mean, and about 99.7% of the data are within 3 standard deviations.

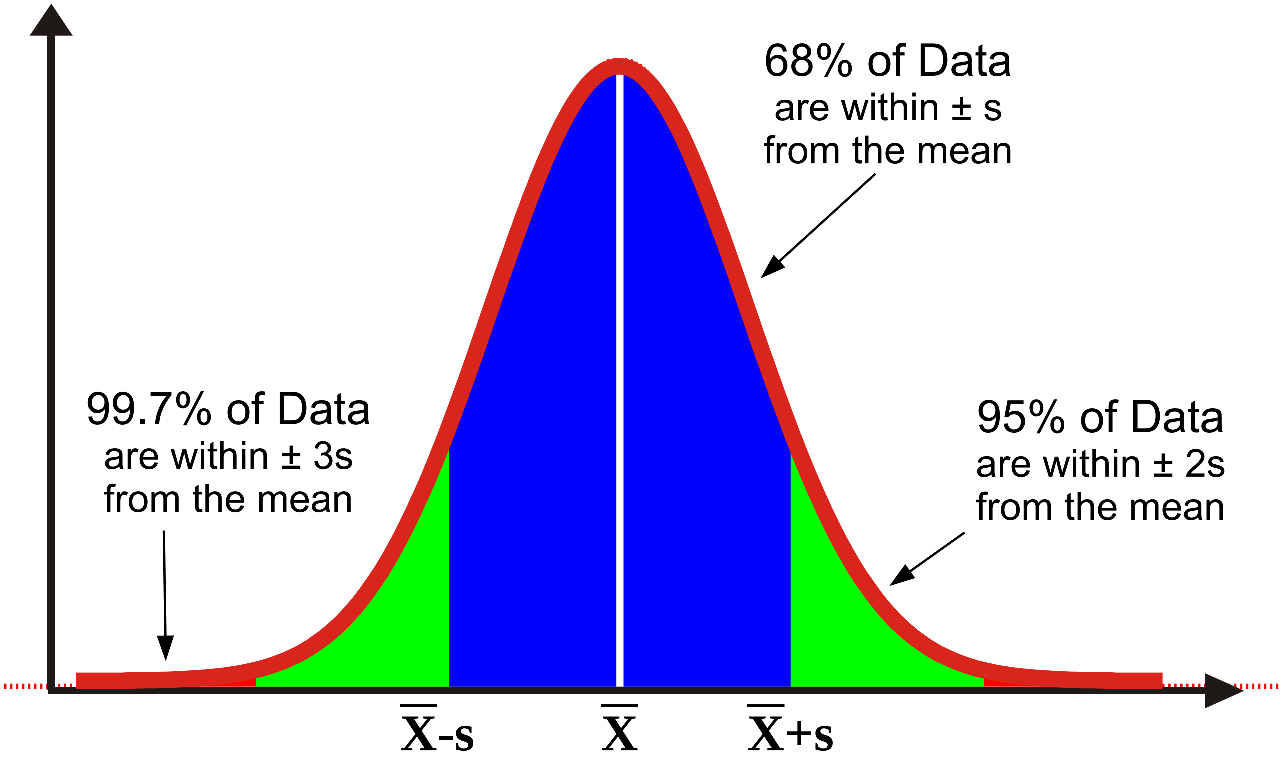


Figure 7.30: If data are distributed Normally, 68% of the values fall within one standard deviation from the mean. 95% of values are within two standard deviations, and 99.7% are within three standard deviations.

If you look at a Normal curve, you can find its standard deviation by locating the places where the curvature changes from positive (concave up) to negative (concave down). Mathematically, these points (called "points of inflection") are where the 2^{nd} derivative of the function is zero. The standard deviation is the horizontal distance from the mean to either of these two points. (See Figure 7.29.)

Exercise 39: It's Only Fitting

We've seen that *gnuplot* can plot data, but it can also plot functions. Several functions, like sin(x), cos(x), and exp(x) are built into *gnuplot*, but you can also define your own functions. Try starting up *gnuplot* and typing the following:

```
p(x) = a*exp(-0.5*(x-m)**2/s**2)
s=2.5
m=35
a=10000
plot "hist.dat" with impulses, p(x)
```

The first line defines a new function `p(x)` that's just the

Normal curve given in Equation 7.4 above. The next three lines set the parameters: `s` is the standard deviation, `m` is the mean ($\overline{X}$), and `a` is the height of the peak.

The last line plots your histogram data from the file `hist.dat` and overlays a Normal curve on top of it. You can see that the shapes are similar, but the curve doesn't exactly match the data.

We could try adjusting the values of `s`, `m`, and `a` by hand to make the curve fit better, but *gnuplot* can do this for us automatically.

Type the following *gnuplot* commands:

```
fit p(x) "hist.dat" via s,m,a
replot
```

The first command tells gnuplot to adjust the parameters `s`, `m`, and `a` to make `p(x)` match the data in `hist.dat`. When it's done, it prints out a lot of information including the new values of the parameters. The second command tells gnuplot to re-do our last graph, which will now draw `p(x)` using the new parameters. Does it fit better now?

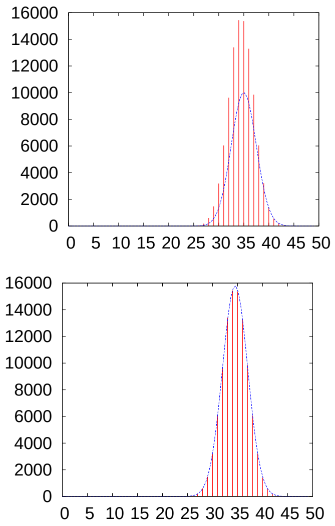


Figure 7.31: A Normal curve superimposed on our `hist.dat` data. The top graph shows a curve that doesn't quite match. The bottom graph shows the curve after we've asked *gnuplot* to adjust the parameters for the best fit.

But what about...?

In the data we've been looking at, each data point is some distance, `d` (positive or negative) from the mean value. The sample standard deviation, `s`, tells us how far a "typical" data point strays from the mean, but there are other ways we could choose to quantify a "typical" deviation. For example, we could look at the average absolute value of d.

The standard deviation has some nice properties, though. In particular, it has a natural relationship to the Normal distribution. As we saw above, 2s is the distance between the "points of inflection" (the places where the curvature goes from positive to negative) of the Normal distribution.

More importantly, statisticians tell us that the sample standard deviation is usually the best estimate of the standard deviation of the infinitely many data points we could possibly collect (the "parent population").

7.8. Exploring The Central Limit Theorem

In Chapter 2 we learned how to simulate rolling dice. For example, Program 2.4 generates a random number between 1 and 6, just like rolling a 6-sided die. Program 7.7, below, is an updated version that rolls a 6-sided die 1,000 times. If we used *gnuplot* to plot this program's output, we would see something like Figure 7.32.

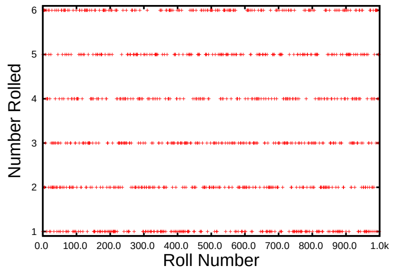


Figure 7.32: The output of `singledie.cpp` plotted using the *gnuplot* command `plot "singledie.dat"`

Notice that we see about the same number of rolls landing on each number, which is what we'd expect from a fair die (or a good random-number generator!). If we made a histogram of the values obtained from rolling a single 6-sided die, it might look like Figure 7.33. As you can see, each value has an equal probability of turning up.

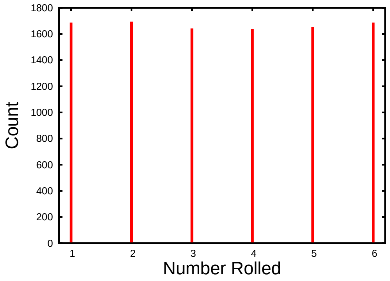


Figure 7.33: A histogram of the values obtained by rolling a single 6-sided die 1,000 times.

Program 7.7: singledie.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
  int roll;
  int min = 1;
  int max = 6;
  int nvals;
  int i;
  double x;

  nvals = max - min + 1;
  srand(time(NULL));

  for ( roll=0; roll<1000; roll++ ) {**
      x = rand()/(1.0 + RAND_MAX);
      i = min + (int)(nvals*x );
    printf( "%d\n", i );
  }
}
```

Some dice games require you to roll two or more dice at once, and add up their numbers. Let's modify Program 7.7 so that it rolls twelve dice at once, instead of just rolling one die. We'll need to add an extra loop and a couple of variables to do that. The result is Program 7.8.

Program 7.8: multidice.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
  int roll, die;
  int min = 1;
  int max = 6;
  int nvals;
  int i, sum;
  double x;

  nvals = max - min + 1;
  srand(time(NULL));

  for ( roll=0; roll<1000; roll++ ) {
    sum = 0;
    for ( die=0; die<12; die++ ) {
      x = rand()/(1.0 + RAND_MAX);
      i = min + (int)(nvals*x );
      sum += i;
    }
    printf ( "%d\n", sum );
  }
}
```

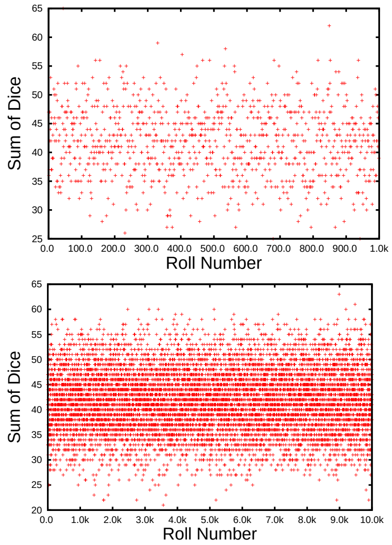


Figure 7.34: The upper figure shows the output of Program 7.8 plotted using *gnuplot*. The bottom figure shows what it would look like if we increased the number of rolls from 1,000 to 10,000.

If we plotted the output of Program 7.8 we'd see something like the upper graph in Figure 7.34. Notice that now the values aren't spread evenly any more. When we roll twelve dice and add them up, their sum is most likely to be somewhere around 42. This is even more apparent in the bottom graph of Figure 7.34, where we've increased the number of rolls to 10,000.

To get a better sense of the distrubution of the values, let's make a histogram of them. We can do that by combining Program 7.8 with Program 7.2. The result is Program 7.9 below. (Notice that we've set the number of dice rolls to 10,000 now.) If we ran this program and plotted its output using the *gnuplot* command

```
plot "dicehist.dat" using 1:4 with impulses"
```

we'd see something like Figure 7.35.

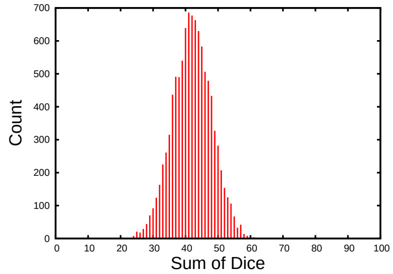


Figure 7.35: A histogram of our dice roll sums, created by Program 7.9, using the following *gnuplot* command:
`plot "dicehist.dat" using 1:4 with impulses`

Program 7.9: dicehist.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
  int roll, die;
  int min = 1;
  int max = 6;
  int nvals;
  int i, sum;
  double x;
  const int nbins = 100;
  int bin[nbins];
  int binno, overunderflow = 0;
  double xlow, xmid, xhi, binwidth;
  double xmin = 0.0;
  double xmax = 100.0;

  binwidth = (xmax-xmin)/nbins;

  for ( i=0; i<nbins; i++ ) {
    bin[i] = 0; // Reset all bins to zero.
  }

  nvals = max - min + 1;
  srand(time(NULL));

  for ( roll=0; roll<10000; roll++ ) {
    sum = 0;
    for ( die=0; die<12; die++ ) {
      x = rand()/(1.0 + RAND_MAX);
      i = min + (int)(nvals*x );
      sum += i;
    }
    binno = (sum-xmin)/binwidth;
    if ( binno < 0 || binno >= nbins ) {
      overunderflow++;
      continue; // Skip this value and jump to the next.
    }
    bin[binno]++; // Increment the appropriate bin.
  }

  for ( i=0; i<nbins; i++ ) {
    xlow = xmin + binwidth*i;
    xmid = xmin + binwidth*(0.5+i);
    xhi = xmin + binwidth*(i+1);
    printf ("%lf %lf %lf %d\n", xlow, xmid, xhi, bin[i]);
  }
  printf ("# Xmin = %lf\n", xmin);
  printf ("# Xmax = %lf\n", xmax);
  printf ("# Binwidth = %lf\n", binwidth);
  printf ("# Nbins = %d\n", nbins);
  printf ("# Saw %d over/underflows\n", overunderflow);
}
```

Figure 7.35 shows that a value of 42 appears almost 700 times when we sum up our twelve dice. The farther away from 42 we get, the less likely we are to see a given sum. The distribution of values looks like a Gaussian or Normal distribution, as described in Section 7.7. As we noted in that section, this effect is known as the "Central Limit Theorem". It tells us that the sum of several random variables tends to take on a Normal distribution.

Even though the distribution of numbers we get from each die is flat, as shown in Figures 7.32 and 7.33, the sum of these numbers approaches a Normal distribution (see Figures 7.34 and 7.35).

The fact that our observed values are centered around 42 makes sense too. Each 6-sided die gives a value between 1 and 6, so the average value we should get from a single roll of a die is $(1+6)/2 = 3.5$. That means that the average value for the sum of twelve dice should be $12 \times 3.5 = 42$.

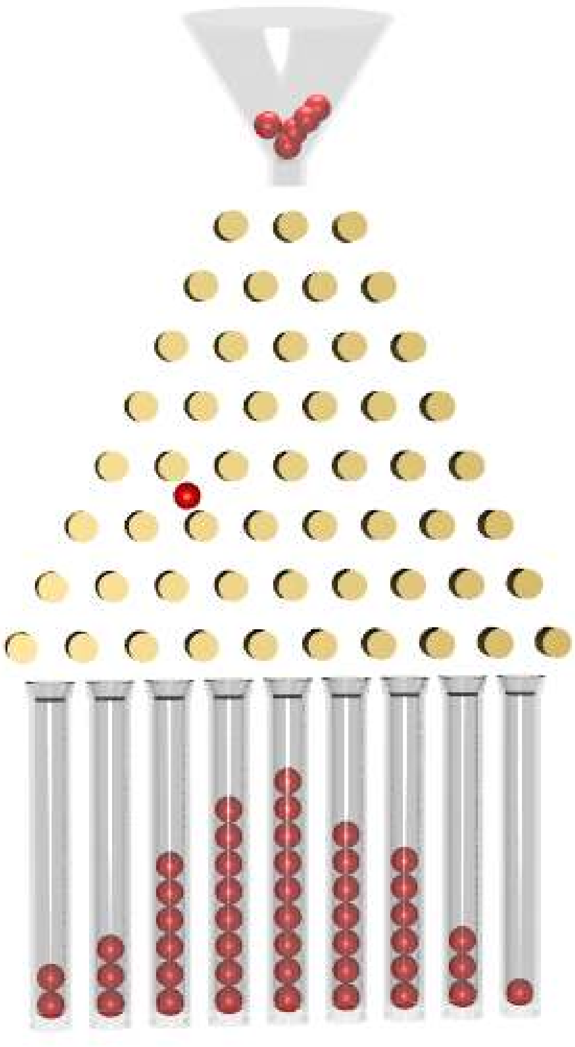


Figure 7.36: Beans bounce off of pegs as they roll down a "Galton Board". At the bottom they fall into bins, like histogram bins. The sum of all the random left and right bounces experienced by the beans results in an approximately Normal distribution.

Source: Wikimedia Commons

7.9. Analyzing Multi-Column Data

Statistics began as a study of demographic data (numerical data about populations), so let's take a look at some "people data" before we finish. The US constitution mandates that a census be taken every ten years, and the task of collecting and analyzing data falls on the US Census Bureau.

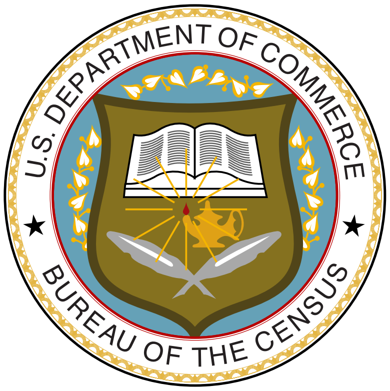


Figure 7.37: The US Census Bureau is charged with conducting a decennial census.

Source: Wikimedia Commons

Census takers collect a lot of data for each household they visit. They might record the number of children, the number of bedrooms in the house, the amount paid monthly in rent, and so forth. We might store the data for each household in a row, with a column for each quantity that was recorded. The result would look something like this:

```
0        1        3    10700        2        2        0
0        1        4     7800        2       40        0
0        1        3    64200        2      130        0
0        1        3       -1     2400      210        0
0        0        1       -1        2       10      780
0        1        3    44600        2       90     1905
...
```

In the following sections, we'll be constructing a program that can read a data file from the US Census Bureau that contains information about 1,285,588 households. The file has seven columns of integers for each household. Each column represents a different measurement:

Column	**Description**
0	Number of related children in household
1	Lot size
2	Number of bedrooms
3	Family income
4	Annual fuel cost
5	Monthly gas cost
6	Monthly rent

Figure 7.38: Taking the census could be a dangerous job. Consider the plight of a census taker asked to survey these denizen's of an 1890 New York "Bandit's Roost". This picture was taken by Jacob Riis, who prowled New York's tenements accompanied by then-Police-Commissioner Theodore Roosevelt, documenting "How the Other Half Lives" (the title of Riis's best-known book).

Source: Wikimedia Commons

The file we've been analyzing, `energy.dat`, contains only one column of data. Only one measurement (the amount of energy deposited) was recorded for each particle that passed through the detector. The census taker, on the other hand, takes several measurements for each family. Let's look at how we might modify our earlier programs to allow them to read such multi-column data.

One way to do it would be to replace our single variable (`energy`, in the earlier programs) with an array. The number of elements in the array will need to match the number of columns in the data file.

Program 7.10 uses this strategy to analyze data from a seven-column data file. In order to read each row, it loops through the seven elements of the array `data`. The new variable `field` specifies which column of the data we want to analyze, and the new program gives the variable `x` the value of `data[field]`. (Program 7.10 sets `field` to 0, but it could be set to any value from 0 to 6.) The new program also changes the name of the data file from `energy.dat` to `census.dat`.

Because Program 7.10 uses a "`for`" loop to read multiple items from each line, we can no longer use a simple `break` when we reach the end of the file, as we did when reading `energy.dat`. Remember from Chapter 4 that the `break` statement only stops the loop it's in. If we used a `break` inside the "`for`" loop of Program 7.10 when we get to the end of the file, the `break` would only stop the "`for`" loop. It wouldn't stop the outer, enclosing "`while`" loop, so the program would keep trying (and failing) to read lines forever.

There are several ways we could handle this. One of them is to use C's "`goto`" statement. A `goto` statement jumps immediately to another location in your program. You might think that this could be a highly dangerous thing to do, and you'd be right. There's a superstition among programmers that says `goto` should never be used, but experts agree[7] that `goto` is sometimes the best solution in one specific case: when your program needs to break out of nested loops like the ones we have in Program 7.10.

Notice the line in Program 7.10 that just says "`done:;`". This is called a "label". A label can be any word, followed by a colon[8], on a line by itself. Labels don't do anything. They just mark a spot in your program. Think of them as bookmarks. When we say `goto done;` we're telling the program to jump to the label named "`done`". When Program 7.10 gets to the end of the file it's reading, the `goto` statement jumps out of the nested loops and continues below the `done:;` label.

Used in this way, `goto` statements can be a safe and efficient way to break out of nested loops. If you think of `goto` as a kind of "super-`break`" it's quite unlikely that you'll be eaten by a velociraptor[9]... but remain vigilant.

[7] See the "exception" under ES.76 in the *CPP Core Guidelines*: https://github.com/isocpp/CppCoreGuidelines.

[8] Notice that this is a colon, not a semicolon. In the examples in this book we'll also put a semicolon after the label, just as we do with other C statements.

[9] See https://xkcd.com/292.

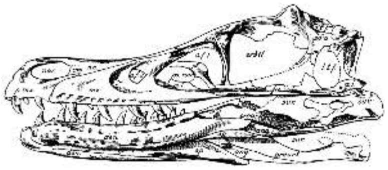


Figure 7.39: Skull of *Velociraptor mongoliensis*.

Source: Wikimedia Commons

Program 7.10: census.cpp

```
#include <stdio.h>
#include <math.h>
int main () {
  int i, binno, overunderflow = 0;
  double x, xlow, xmid, xhi, binwidth;
  double xmin = 0.0;
  double xmax = 50.0;
  const int nbins = 50;
  int bin[nbins];
  double sum = 0.0;
  double sum2 = 0.0;
  int nvalues = 0;
  FILE *input;
  int field=0; // Select column 0 from data.
  double data[7]; // Add "data" array.

  binwidth = (xmax-xmin)/nbins;

  for ( i=0; i<nbins; i++ ) {
    bin[i] = 0; // Reset all bins to zero.
  }

  input = fopen( "census.dat", "r" );
  while ( 1 ) {
               for ( i=0; i<7; i++ ) {
                if ( fscanf( input, "%lf", &data[i] ) != 1 ) {
                  goto done;
                }
               }

              x = data[field]; // Choose which column.

              sum += x;
              sum2 += pow( x, 2 );
              nvalues++;

              binno = (x-xmin)/binwidth;
              if ( binno < 0 || binno >= nbins ) {
                overunderflow++;
                continue;
              }
              bin[binno]++;
  }
 done:;
  fclose(input);

  for ( i=0; i<nbins; i++ ) {
    xlow = xmin + binwidth*i;
    xmid = xmin + binwidth*(0.5+i);
    xhi = xmin + binwidth*(i+1);
    printf ("%lf %lf %lf %d\n", xlow, xmid, xhi, bin[i]);
  }
  printf ("# Field number %d\n", field);
  printf ("# Xmin = %lf\n", xmin);
  printf ("# Xmax = %lf\n", xmax);
  printf ("# Binwidth = %lf\n", binwidth);
  printf ("# Nbins = %d\n", nbins);
  printf ("# Saw %d over/underflows\n", overunderflow);
  printf ("# Mean value is %lf\n", sum/nvalues );
  printf ("# Std. dev. is %lf\n",
          sqrt( (sum2 - sum*sum/nvalues)/(nvalues-1) ) );
  printf ("# Nvalues = %d\n", nvalues );
}
```

Get 7 items from each line

Read lines from file

Jump out of nested for and while loops when we reach the end of the file

7.10. Filtering Data

Census takers can't always collect all measurements from every household. Sometimes a measurement just doesn't apply. What's the monthly rent on a house that's not being rented? What's the annual household income for an unoccupied house? Our data sets will sometimes contain special values that indicate "Not Applicable". We might not want to include these values in our averages, or show them on our histograms. We could think of this a "filtering" our data.

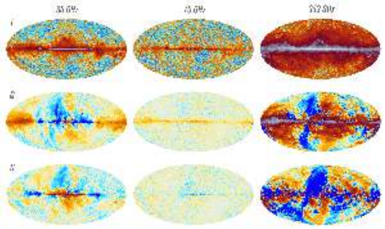


Figure 7.40: The *planck* spacecraft examined the microwave radition leftover from the Big Bang. The figure above shows analyses of *planck's* data with several different filters applied.
Source: *Planck Mission, European Space Agency*

In the census data we're going to look at, these special values are indicated by zeros or negative numbers. By making a couple of changes, we can cause our program to ignore such values. First, we want to look for special values whenever we read a line from our data file. When we find one, we want to skip that line and just go on to the next. We can accomplish this by adding the following section before the "`sum +=`" in Program 7.10:

```
if ( x <= 0 ) {
  continue; // Ignore zeros and negatives.
}
```

We'll probably want to know how many values were ignored (or, equivalently, how many *weren't*). It would be a good idea to add a line like the following at the end of the program, along with the other numbers we print out:

```
printf ("# Saw %d data values\n", nvalues);
```

The variable `nvalues` tells us how many data points we really analyzed, not counting those we filtered out.

We can modify our data analysis program to filter our data any way we like. We might even look at the other columns on each line when deciding whether or not to use the data on that line. For example, maybe we're interested in the number of children per household, but only want to look at families paying more than $500 per month in rent.

7.11. Setting Analysis Parameters

Program 7.10 explicitly chooses a particular column to analyze by setting the `field` variable. It would be nice if the program asked us which column we wanted to use. We can easily add a section somewhere before our `while` loop to do this:

```
printf ( "Pick a column [0-6]: " );
scanf ( "%d", &field );
```

If we pick a different column, we might also want to use a different bin width. (This is the width of the bins into which we drop our "virtual marbles" while making a histogram.) A bin width of 1 is fine if we're looking at the number of children per household, but we might want a width of 10,000 if we're looking at annual household income. An income difference of $1 isn't very interesting, but $10,000 would be. We could add another section to our program for setting the bin width:

```
printf ( "Enter bin width: " );
scanf ( "%lf", &binwidth );
```

If we specify `binwidth`, we can calculate the value of `xmax` (the maximum value we're interested in) like this:

```
xmax = binwidth*nbins + xmin;
```

Let's leave the lower end of our range (`xmin`) at zero, since the data in each colum of our data set includes some small values.

We could add any number of similar sections to the beginning of a data analysis program, to allow us to set any parameters we need. Maybe we want to analyze only data for households with annual incomes in a given range (say, between $20,000 and $30,000). In that case, the program could ask for `minincome` and `maxincome`, and use those variables when filtering the data.

Figure 7.41: Some members of the author's family, *circa* 1939.

7.12. Using stderr

If our program asks the user for parameters, we introduce another complication: some of the program's output (the request to "Enter bin width", for example) needs to go to the computer's display, so the user can see it, but other output (the histogram data) needs to be written into a file so we can plot it with *gnuplot*. If we just type `./census > output.dat` then the user won't see the requests for entering parameters, and the program will just sit forever waiting for them.

There are several ways to solve this problem. For one, we could use `fprintf` to write the histogram into a file instead of sending it to the display, as we saw in Chapter 5.

Let's look at another way of doing it, though. As we saw in Chapter 5, we can open a file with `fopen` like this:

```
FILE *output;
output = fopen("output.dat","w");
```

The variable `output` is a "file handle" that we can use later with `fprintf`. We can open as many files as we want, and choose which file handle to use when we want to print something into one of them.

It turns out that three file handles are automatically created whenever you run your program. These are named `stdout`, `stderr`, and `stdin`. The `stdout` file handle doesn't point to a real file. Instead, it points to your display. The `printf` statement uses this file handle whenever it prints something. The statement `printf("Hi!");` is just equivalent to `fprintf(stdout,"Hi!");`.

When you type a command like "`./census > output.dat`" the computer disconnects `stdout` from your display and connects it to the file `output.dat` instead. This makes the output of any `printf` statements go into the file instead of to your screen.

The `stdin` file handle points to your keyboard. The statement `scanf("%d", &i);` is the same as `fscanf(stdin,"%d", &i);`.

The third predefined file handle, `stderr`, also points to your display, but it's intended to be used for errors and warnings. Imagine, for example, that you've typed "`./census > output.dat`" but your program crashes with a Segmentation Fault error. The error message should be sent to your display, not to the file. Error messages like this are sent to `stderr`, which is still connected to your display.

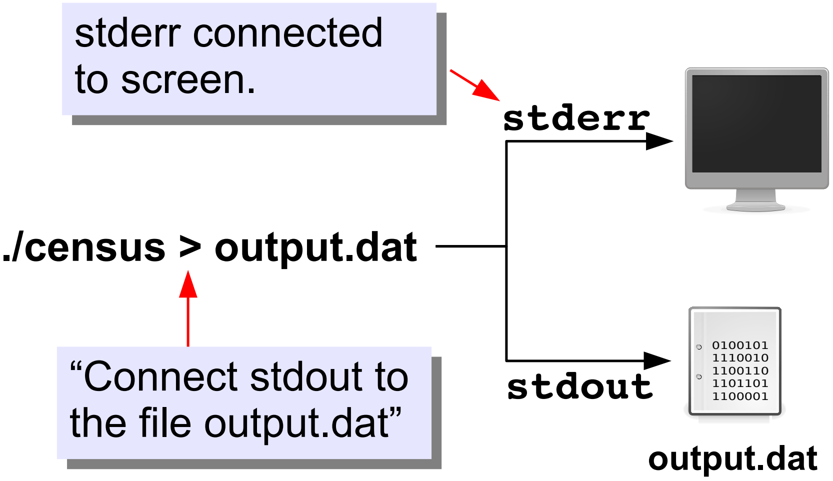


Figure 7.42: The predefined file handles `stdout` and `stderr` both start out pointing at your display, but they can be redirected elsewhere.

We can use `stderr` for our own purposes, too. We want our “Enter bin width” message to go to the display even if we’ve redirected the program’s output into a file. All we need to do is send those messages to `stderr` instead of `stdout`. We can do that by modifying a couple of `printf` statements:

```
fprintf ( stderr, "Pick a column [0-6]: " );
scanf ( "%d", &field );

fprintf ( stderr, "Enter bin width: " );
scanf ( "%lf", &binwidth );
```

Instead of `printf`, we use `fprintf` to send these messages to `stderr`.

But what about. . . ?

Are there other ways we could split the program’s output between display and file? Why yes, I’m glad you asked!

One way involves the third predefined file handle, `stdin`. This normally points to your keyboard, and it’s used by `scanf` whenever it reads some input. However, just like `stdout`, you can disconnect `stdin` from the keyboard and connect it to a file instead. If you did that, you could cause your program to read stored answers from a file, rather than having to type them in at the keyboard. Figure 7.43 shows how to do this, using the “<” symbol on the command line. If we did it this way, the program would expect to find two numbers in the file `input.dat`: the column number we want to analyze, and the bin width.

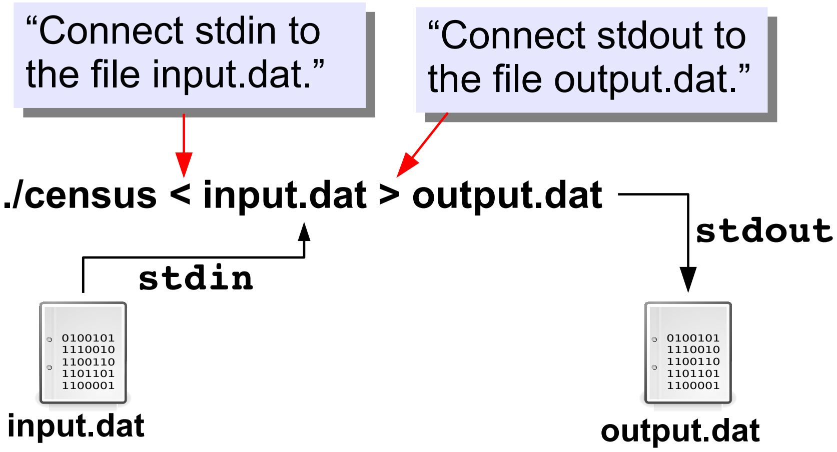

"Connect stdin to the file input.dat."
"Connect stdout to the file output.dat."
./census < input.dat > output.dat
stdout
stdin
0100101
1110010
1100110
1101101
1100001
input.dat
0100101
1110010
1100110
1101101
1100001
output.dat

Figure 7.43: We could redirect both `stdout` and stdin if we wanted to. The "<" on the command line means "Read input from this file", just as the ">" means "Write output to this file". If the program asks us some questions, we can save our answers in the file `input.dat`. The program will read them from there, instead of waiting for us to type them.

7.13. Improved Analysis Program

Program 7.11 is an improved analysis program that incorporates all of the improvements we've talked about in the preceding sections. When we run the program it asks us which column (0 through 6) we want to analyze, then it asks us what bin width we want to use. The histogram data is sent to the display, unless we redirect it to a file.

Figure 7.44 shows some results from the program. To plot the income graph, for example, we did this:

```
./census > income.dat
```

and then answered the questions:

```
Pick a column [0-6]: 3
Enter bin width: 10000
```

The output file was graphed with *gnuplot* as with our earlier histograms.

Notice that the data here aren't bunched into Normal distributions like the energy data. In the energy case, we were making many measurements of the same value (the energy of some kind of particle striking our detector). The only variations in these measurements were due to random factors.

The census data, on the other hand, is inherently different from one household to another. The distribution of values could give us some real information about people's lives. Nonetheless, we can still calculate the mean values of things like income, and calculate the standard deviation of our data sample. The standard deviation still tells us something

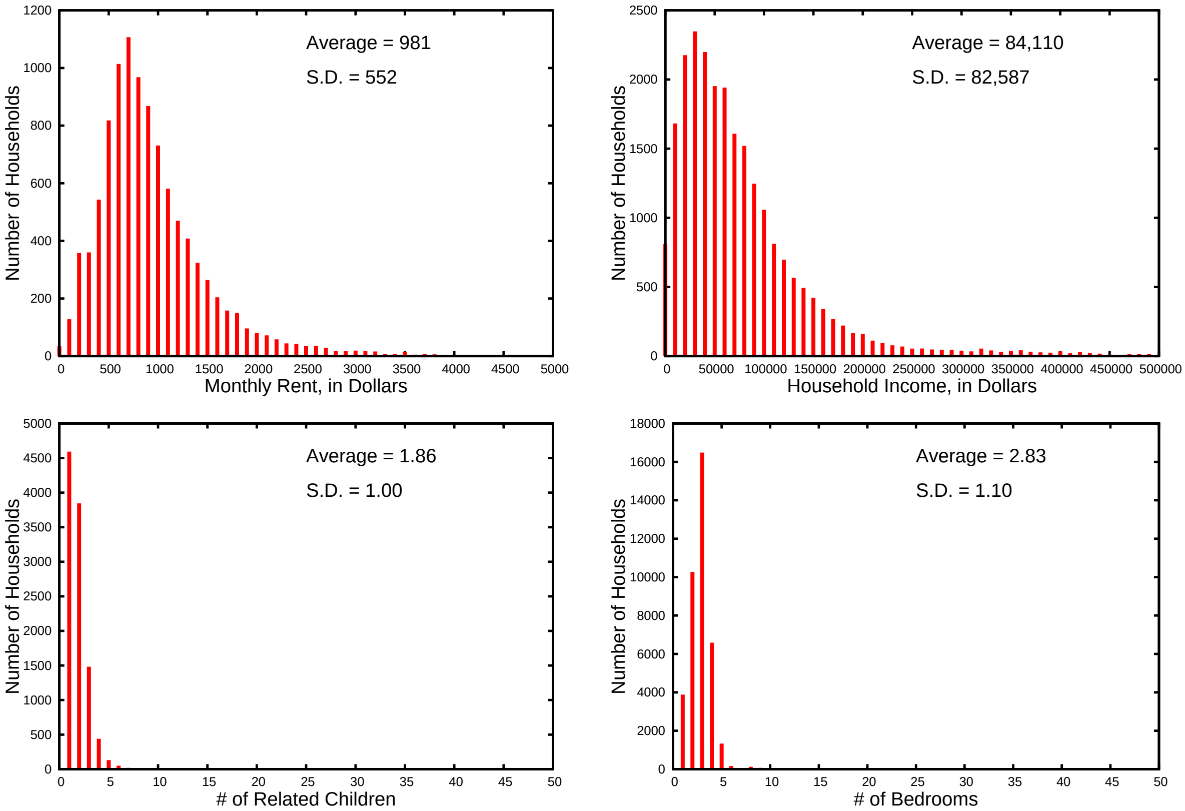


Figure 7.44: Some results from Program 7.11, plotted with *gnuplot*. Bin width was set to 10,000 for the income graph, 100 for the rent graph, and 1 for the bedrooms and children graphs.

about the width of the distribution, as it did with the energy data, even though the income distribution is far from Normally-distributed.

Exercise 40: Little Pink Houses

For this exercise you'll need a copy of the file `census.dat`. You'll find instructions for obtaining it in Appendix C.3 on page 624.

First, examine `census.dat` with *gnuplot*. Start *gnuplot* and type the command:

```
plot "census.dat" using 4
```

gnuplot numbers columns starting with 1, so this should display a graph of household income similar to Figure 7.45. Note the bar of negative values representing special cases that our analysis program will ignore.

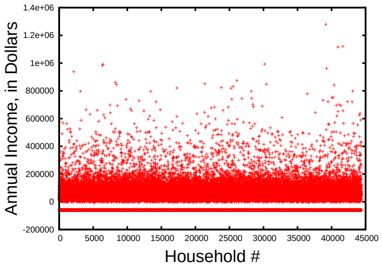


Figure 7.45: The output of the *gnuplot* command `plot "census.dat" using 4`.

Now exit from *gnuplot* and compile Program 7.11 (the new `census.cpp`, on Page 254). Run the program like this:

```
./census > income.dat
```

Select the income by choosing column number 3 (the program starts numbering the columns with 0). Use a bin width of ten thousand.

Now start up *gnuplot* again and ask it to plot the results of your analysis:

```
plot "income.dat" using 1:4 with boxes
```

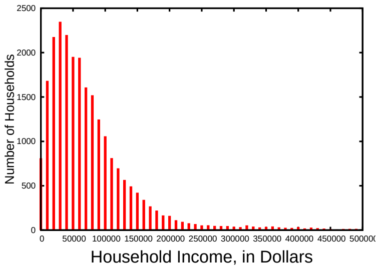


Figure 7.46: The income histogram produced by our analysis program.

By saying "`using 1:4`" we tell gnuplot to use column 1 (the smallest value in each bin) as the value on the x axis, and column 4 (the number of "virtual marbles" in each bin) as the y value. The graph shows us how many households are in each income range.

If you have time, try plotting other columns from the `census.dat` file and analyzing them.

Program 7.11: census.cpp, Version 2

```
#include <stdio.h>
#include <math.h>
int main () {
  int i, binno, overunderflow = 0;
  double x, xlow, xmid, xhi, binwidth;
  double xmin = 0;
  double xmax;
  const int nbins = 50;
  int bin[nbins];
  double sum = 0.0;
  double sum2 = 0.0;
  int nvalues = 0;
  FILE *input;
  int field=0;
  double data[7]; // Add "data" array.

  fprintf ( stderr, "Pick a column [0-6]: " );
  scanf ( "%d", &field );

  fprintf ( stderr, "Enter binwidth: " );
  scanf ( "%lf", &binwidth );

  xmax = binwidth*nbins + xmin; // Calculate xmax from xmin and binwidth.

  for ( i=0; i<nbins; i++ ) {
    bin[i] = 0; // Reset all bins to zero.
  }

  input = fopen( "census.dat", "r" );
  while ( 1 ) {
    for ( i=0; i<7; i++ ) {
      if ( fscanf( input, "%lf", &data[i] ) != 1 ) {
        goto done;
      }
    }

    x = data[field]; // Choose which column.

    if ( x <= 0 ) {
      continue; // Ignore zeros and negatives, since they're special.
    }

    sum += x;
    sum2 += pow( x, 2 );
    nvalues++;

    binno = (x-xmin)/binwidth;
    if ( binno < 0 || binno >= nbins ) {
      overunderflow++;
      continue; // Skip this value and jump to the next.
    }
    bin[binno]++; // Increment the appropriate bin.
  }
 done:;
  fclose(input);

  for ( i=0; i<nbins; i++ ) {
    xlow = xmin + binwidth*i;
    xmid = xmin + binwidth*(0.5+i);
    xhi = xmin + binwidth*(i+1);
    printf ("%lf %lf %lf %d\n", xlow, xmid, xhi, bin[i]);
  }
  printf ("# Field number %d\n", field);
  printf ("# Xmin = %lf\n", xmin);
  printf ("# Xmax = %lf\n", xmax);
  printf ("# Binwidth = %lf\n", binwidth);
  printf ("# Nbins = %d\n", nbins);
  printf ("# Saw %d over/underflows\n", overunderflow);
  printf ("# Mean value is %lf\n", sum/nvalues );
  printf ("# Std. dev. is %lf\n",
          sqrt( (sum2 - sum*sum/nvalues)/(nvalues-1) ) );
  printf ("# Nvalues = %d\n", nvalues );
}
```

7.14. Conclusion

In this chapter we've looked at some basic techniques for doing statistical analysis of data with computer programs. Histograms and calculations of the mean and standard deviation are primary tools for data analysis in the sciences.

The details can vary greatly, but the outline of most data analysis programs will look much like Figure 7.47. We've discussed each of these steps as we developed and improved our census analysis program.

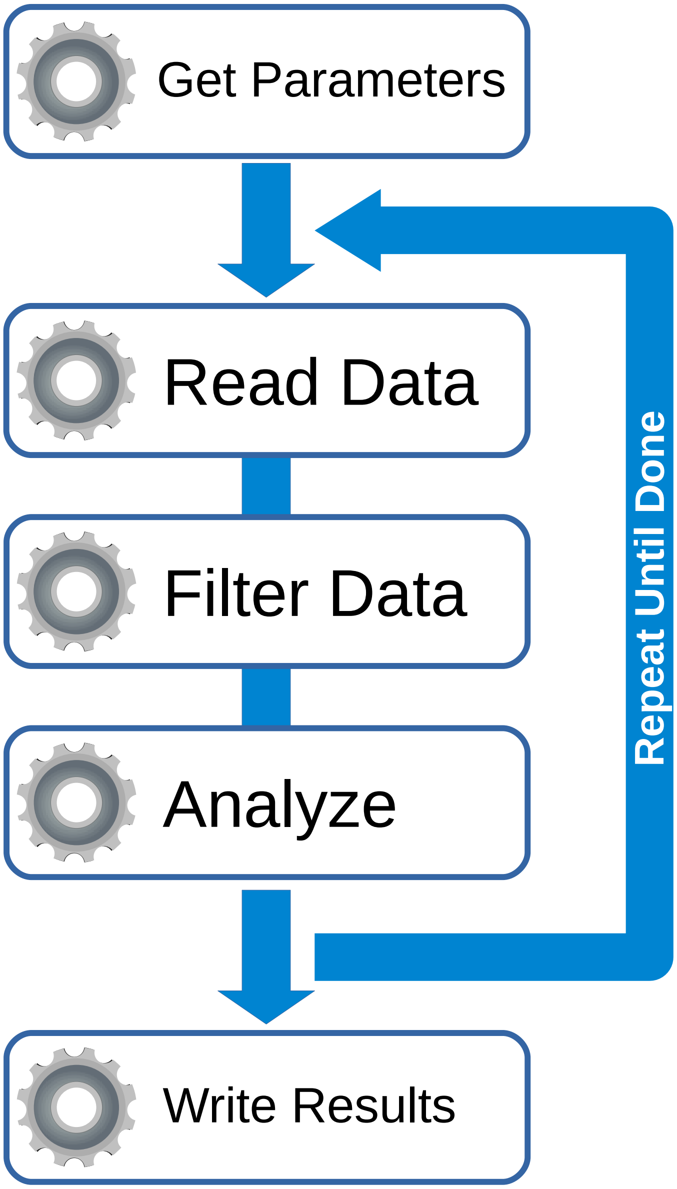


Figure 7.47: The figure above shows an outline of a typical data analysis program.

Practice Problems

1. Write a small program named `listmean.cpp` that finds the mean value of a list of numbers. Start out with an array of numbers, like this:

   ```
   double x[10] = {0,1,2,3,4,5,6,7,8,9};
   ```

   Use a "`for`" loop to go through the elements of the array, adding them up. At the end of the program, print out the mean value of these numbers.

2. The "mean" that we've talked about in this chapter is the "arithmetic mean". There are other kinds of mean value that we could calculate. One of them is called the "geometric mean". To find the geometric mean of a set of numbers, multiply them together and take the n-th root of their product, where n is how many numbers are in the set. For example, if we have the numbers 4, 5, and 6, their geometric mean would be:

   $$\sqrt[3]{4\times5\times6} \quad \text{or, alternatively} \quad (4\times5\times6)^{1/3}$$

   Write a program named `geomean.cpp` that calculates the geometric mean of these nine numbers:

   ```
   double x[9] = {1,2,3,4,5,6,7,8,9};
   ```

   Hints:

   - You can use the `pow` function to find the n-th root. For example, the 4^{th} root of 38 would be `pow( 38, 1.0/4 )`. Note that it's important to say `1.0/4` instead of `1/4`, because the latter would tell the computer that you wanted to trim the decimal places off of the result.
   - When summing up a bunch of numbers we start with `sum = 0.0` and add each number by saying `sum += x`. When multiplying a bunch of numbers, you might start by saying `product = 1.0`, then multiply by each number by saying `product *= x`.

3. When you roll two six-sided dice the sum of their values can be any number between 2 and 12. We know that some combinations are more unlikely than others. For example, you're less likely to roll a 12 than a 7. Let's see if we can quantify how likely each possible combination is.

   Write a program named `2dice.cpp` that rolls two dice 1,000,000 times and counts how many times each possible sum occurs. Each time we roll, we'll get two random numbers between 1 and 6. If you have an `int` variable named `die1` for the first die, you can generate a random value for it like this:

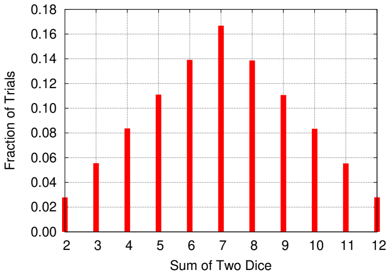


Figure 7.48: How often do you roll a particular sum? It looks like 7 comes up most often – about 17% of the time, and 2 and 12 are least common – about 3% each.

```
die1 = 1 + 6.0*rand()/(1.0+RAND_MAX);
```

To record the results of the rolls, make a histogram using a 13-element integer array like this:

```
int bin[13];
```

Make sure you set all of the elements of this array to zero at the beginning of the program. Use the elements of the array to record how many times you saw a particular sum. For example, `bin[7]` will hold the number of times you rolled a 7. (Notice that we won't use `bin[0]` or `bin[1]` since we can never get a sum of zero or one, but that's OK. We'll just ignore those bins.)

After you've done all the rolls, your program should have a loop that prints the contents of elements 2 through 12 of `bin`. The format should be:

```
printf( "%d %d %lf\n", i, bin[i], bin[i]/1000000.0 );
```

That will print the bin index, the contents of that bin, and the fraction of our 1,000,000 rolls that ended up with that sum. If you run the program like "`./2dice > 2dice.dat`" and then plot the output using *gnuplot* you should see something like Figure 7.48.

4. Using Program 7.5 as a starting point, create a program called `calcstats.cpp` that prompts the user to enter numbers, one at a time, and then prints out the mean value and standard deviation of the numbers entered.Make sure the program can accept numbers that have decimal places.

   You'll need to think about how the user can let the program know that he/she is finished entering numbers. If you only allow positive numbers, you could ask the user to enter "`-1`" to stop the program, but then you'd be unable to enter -1 as a number! There's a better way to do it. Remember that `scanf` and `fscanf` return a value that tells you how many numbers were successfully read. That means that if you use `scanf` in a "`while`" loop just like we used `fscanf` in Program 7.1, the loop will stop if the user enters anything that's not a number[10]. Use this trick in your program. **Hint:** You won't need to open or close any files. Just use `scanf` in the way we've been using `fscanf` in programs like Program 7.1.

My first calculator was one of these. It was a present from my Dad in the early 1970s. It was also the first calculator Texas Instruments made. Being a kid who struggled with long division, I loved this device!

Source: *Wikimedia Commons*

5. Imagine an inebriated person standing beside a lamppost. He wants to get home, so he starts walking, but each time he takes a step it's in a different, random, direction. How far away from the lamppost will he be, on average, after 100 steps?

   This is a well-known problem in mathematics called "the drunkard's

[10] The usual convention is to ask the user to press Ctrl-D when they're done entering data. That's a special character that represents the "End Of File" (EOF).

walk". As you can see from Figure 7.49, the distance travelled by the drunkard can vary a lot from one trial to the next. If he walked in a straight line, he'd end up 100 steps away from the lamppost, but most of these random paths leave him much closer.

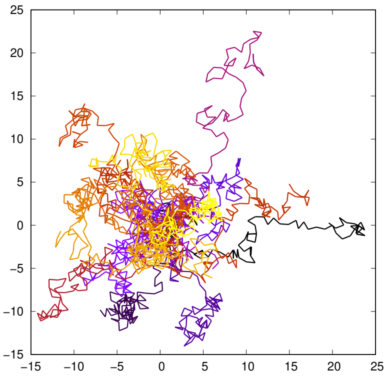


Figure 7.49: The paths of 20 drunken people, each shown in a different color. The lamppost is at the origin. The distance units are "steps", which we assume to be of equal length. Each person has taken 100 steps.

Write a program named randomwalk.cpp that simulates 1,000 of these 100-step paths and prints out the average final distance from the lamppost. (Measure all distances in "steps", which we assume to be of equal length.) Make sure you use srand(time(NULL)) to choose a different "seed" for the random number generator each time you run your program.

Here are a few hints to help you:

- You'll need a pair of nested loops: An outer loop for each path, and an inner one for each step.
- Keep track of the person's position with a couple of variables, x and y. Remember to set them both back to zero at the beginning of each path.
- Every time the person takes a step, generate a random angle like this:

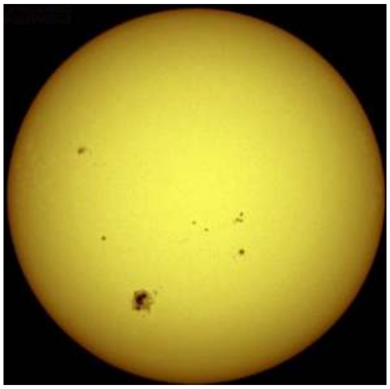
  ```
  angle = 2.0*M_PI*rand()/(1.0+RAND_MAX);
  ```

  then add cos(angle) to x and sin(angle) to y to get the person's new position.
- At the end of each path, calculate the final distance from the origin like this:

  ```
  distance = sqrt( x*x + y*y );
  ```

  and add that to a sum of all of the distances, for use later when you compute the mean distance.
- To check your work: your program should find that the average final distance is about 8.86 steps. This is 0.886 × the square root of the number of steps.

This kind of random motion is common in nature, making the drunkard's walk an important problem in science. In physics, for example, it describes the random motions of molecules in a gas, or the motion of impurities jumping across a surface. In chemistry it describes the shapes of polymers. In economics, random walks can even explain some of the variation in stock prices.



Photons generated in the center of the sun follow a "drunkard's walk" path as they make their way to the sun's surface. This twisty path can include trillions of steps and take as much as a million years to complete.

Source: *Wikimedia Commons*

6. Modify Program 7.11 so that it asks the user for two new parameters:

maxincome and minincome (maximum and minimum income) as described on Page 248. Use these in the filter section of the program (the section where we currently check to see if x is less than or equal to zero). Skip the current row of data if the following is true:

```
data[3] < minincome || data[3] > maxincome
```

7. Write a program named liststats.cpp that reads a list of numbers, one number per line, from a file named liststats.dat and prints how many numbers were read, their mean, and their standard deviation.

   You can test your program with data generated by the three small programs at the end of this chapter: urand, nrand, and prand. For example, to use nrand to create a liststats.dat file, you could type:

   ```
   ./nrand > liststats.dat
   ```

   You should find that the numbers generated by nrand have a mean of about zero, and a standard deviation of around 1. The numbers generated by prand should have a mean of about 3 and a standard deviation of about $\sqrt{3}$.

8. Write a program named arraystats.cpp that defines an array of numbers like this:

   ```
   const int nvalues = 10;
   double x[nvalues] = {0,1,2,3,4,5,6,7,8,9};
   ```

   Have the program calculate and print the mean and the standard deviation of these numbers.

9. Write a program named bufferstats.cpp that defines an array of numbers like this:

   ```
   const int n = 1000;
   double x[n];
   ```

   Have the program open a file named bufferstats.dat that will contain one number per line. Use a "for" loop to read the first 1,000 numbers from the file, and put them into the 1,000 elements of the array x. Close the file, then have a second "for" loop that calculates the mean and standard deviation of the numbers. Print these values at the end of the program, in a friendly easy-to-read format.

You can test your program with data generated by the three small programs at the end of this chapter: `urand`, `nrand`, and `prand`. For example, to use `nrand` to create a `bufferstats.dat` file, you could type:

```
./nrand > bufferstats.dat
```

You should find that the numbers generated by `nrand` have a mean of about zero, and a standard deviation of around 1. The numbers generated by `prand` should have a mean of about 3 and a standard deviation of about $\sqrt{3}$.

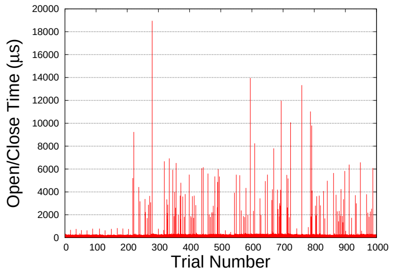


Figure 7.50: If you graphed the numbers from the `jitter` program, they might look like this. As you can see, sometimes an open/close takes a lot longer than usual.

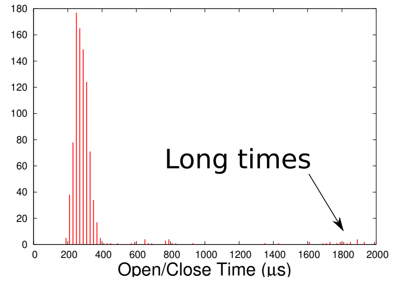


Figure 7.51: If you made a histogram from the numbers, it might look like this. Note that most of the data are clustered around 300 microseconds here, but there are some measurements that go all the way up to thousands of microseconds. (This graph throws away anything bigger than 2,000 μs.)

10. The following program tests how fast your computer can create files. The program repeatedly opens a file ("jittertest.dat"), writes into it, then closes it. As it's doing this it keeps track of how long each open/write/close cycle takes (in microseconds).

Program 7.12: jitter.cpp

```
#include <stdio.h>
#include <sys/time.h>
#include <math.h>
long epoch;
void startclock(){
  struct timeval t;
  gettimeofday(&t, NULL);
  epoch = t.tv_sec * (int)1e6 + t.tv_usec;
}
int microtime(){
  struct timeval t;
  gettimeofday(&t, NULL);
  return( (int)(t.tv_sec * (int)1e6 +
        t.tv_usec - epoch) );
}

int main () {
  int i;
  int tstart, delay;
  FILE * output;

  startclock();

  for ( i=0; i<1000; i++ ) {
    tstart = microtime();
    output = fopen( "jittertest.dat", "w" );
    fprintf( output, "Testing...\n" );
    fclose( output );
    delay = microtime() - tstart;
    printf ( "%d\n", delay );
  }
}
```

The top part of the program (everything above `int main()`) is just some magic that lets us measure time to microsecond accuracy. Some of this will become clear in Chapters 9 and 12, but for now, don't worry about how it works.

The program's "`for`" loop opens, writes, and closes a file 1,000 times. Before opening the file, the program saves the current time (in microseconds) in the variable `tstart`. After the file is closed, the program looks at the new time and calculates how long it took to open, write, and close the file. This time (again in microseconds) is stored in the variable named `delay` and printed with `printf`.

Copy this program, compile it and run it. You should see a string of mostly 3-digit numbers. Now modify the program so that it calculates the mean and standard deviation of `delay` and prints those values at the end of the program.

The mean value will tell you how long, on average, it takes your computer to open a file, write a little text into it, and close the file.

11. Imagine that you're a scientist doing an experiment. You measure the velocity of a dropped ball at several times during its fall. Being a good scientist, you'll probably repeat the experiment several times. Your measurements will be a little different every time because of random things like jiggling your experimental equipment or air currents in the room. By looking at these differences you can get an idea of the uncertainty in your measurements.

t	v	s
0.00	0.00	0.5
1.00	14.88	3.1
2.00	23.08	4.4
3.00	18.04	5.4
4.00	32.78	6.2
5.00	54.27	7.0
6.00	59.86	7.6
7.00	58.04	8.2
8.00	79.84	8.8
9.00	110.69	9.3

Figure 7.52: This table shows the ball's average velocity (v) at several different times during its fall, along with the standard deviation (s) of the velocity at that time.

The data you collect might look like Figure 7.52. If you graph your data, it might look like Figure 7.53. The bars above and below each point represent the uncertainties in the measurements of the velocity. Their length is the standard deviation of all the velocity measurements made at that point.

It looks like the data points might approximately lie on a straight line, like the one drawn in Figure 7.53, but we'd like to have some kind of numerical value that tells us how well these points match the line.

One way to measure the goodness of such a match is a quantity called *chi-squared* (χ^2). Chi-squared measures how well a set of data matches a model (in this case, a particular straight line). For our velocity data, we could calculate chi-squared like this:

$$\chi^2 = \sum_{i=1}^{N} \frac{(v_i - v_{line})^2}{s_i^2} \tag{7.5}$$

where N is the number of data points, v_i and s_i are the measured velocity and standard deviation values, and v_{line} is the velocity predicted by the straight line. The χ^2 value compares the differences

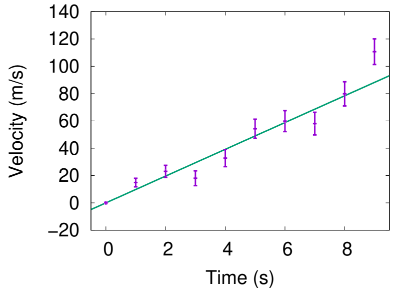


Figure 7.53: A graph of velocity *versus* time measurements, with error bars.

between measured and predicted values with our experimental uncertainties.

Since the sum in Equation 7.5 adds up the squares of a bunch of numbers, it gets bigger as we add more and more data points. To make it easier to judge whether a chi-squared value represents a good match between data and model, we often use a related quantity called the *reduced chi-squared*. It's defined like this[11]:

$$\chi^2_{red} = \frac{\chi^2}{N} \tag{7.6}$$

If a model is a good match to a set of data, the reduced chi-squared should be close to 1.

Copy the three columns of numbers from Figure 7.52 into a file named `velocity.dat`, then write a program name `chi.cpp` that reads data from this file and compares it to a straight line by calculating χ^2. The program should start proceed like this:

- Use a variable named `chisq` to hold the value of X^2. Set this to zero initially.
- Use a variable named `n` to count the number of data points. Set this to zero initially.
- Start by asking the user for the slope and y-intercept of the line they want to test.
- Open `velicity.dat` for reading.
- Use a `while` loop to read values for `t`, `v`, and `s` from each line of the file.
- Each time you read a line, use the `t` value and the slope and y-intercept to calculate the velocity value predicted by the straight line:

  ```
  vline = yint + slope*t;
  ```
- Add $(v_i - v_{line})^2/s_i^2$ to the value of χ^2.
- Add one to the value of `n`.
- Keep doing this until you've read all of the data from the file.
- After reading all of the data, calculate the reduced chi-squared (χ^2_{red}).
- Print the values of χ^2 and χ^2_{red}.

Test your program by entering a slope of 9.8 and a y-intercept of zero. The reduced chi-squared should be on the order of 1, showing

[11] If we plan on searching for the model that best matches some data, we'll usually want to divide by a number slightly smaller than N (we'd use N-2 for finding the best straight line, for example), but that's beyond the scope of what we can describe here.

that this is a reasonable fit to the data. If you try a slope of 5 or 15, for example, you should get a much larger value for χ^2_{red}.

12. The mean tells us where the center of our data lies, and the standard deviation tells us how spread out the data are, but these aren't the only quantitative measurements we can derive from a list of numbers. Another thing we can measure is the *skew* of the numbers. This tells us how symmetrical they are about the mean. Data like those shown in Figures 7.54 and 7.55 have a small skew, because the numbers are symmetrically distributed on either side of the mean. Data like those shown in Figure 7.56 have a large skew, because the distribution is asymmetric: it's *skewed* to one side.

    The skew of a set of N numbers X_i is defined as:

    $$g = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{X_i - \overline{X}}{s}\right)^3 \tag{7.7}$$

    where $\overline{X}$ is the mean value and s is the standard deviation. Notice that we first need to calculate $\overline{X}$ and s before we can start calculating the skew. This means that any program we write to calculate the skew will need to look at all of the numbers twice: once to calculate the mean and standard deviation, and then again to calculate the skew.

    Skew is a little more slippery than mean and standard deviation. Small random variations in the data can cause it to appear to be skewed, even though it isn't really. One way to gauge the significance of a skew value is to compare it to the expected random variation of the skew of a normal distribution, which is

    $$\sigma_{ns} = \sqrt{15/N} \tag{7.8}$$

    If the skew value you calculate is several times larger than this, then it's probably significant. If it's smaller, then it's probably just random wiggling.

    Write a program named `skew.cpp` that does the following:

    - Open, for reading, a file named `skew-data.dat` containing a single column of numbers with decimal places.
    - Read the data from the file and calculate the mean and standard deviation of the numbers.
    - Now close the file.

- Re-open the data file for reading. This will cause your program to start back at the beginning of the file.
- Read the data from the file and use Equation 7.7 to calculate the skew of the numbers. **Hint:** Use a variable named `sum3` to accumulate the sum of the cubes shown in this equation, then after adding up all of these, divide `sum3` by N to find the skew.
- At the end, print the number of values that were in the file, the mean, the standard deviation, and the skew (g in Equation 7.7 above). Also print σ_{ns}, defined in Equation 7.8 above, and print the ratio g/σ_{ns}.

In order to test your program you'll need a data file for it to read. At the end of this chapter you'll find three small programs that can be used to generate different kinds of data sets. They're named `urand`, `nrand`, and `prand`. You can use one of them to make a `skew-data.dat` file for your program to read. For example:

```
./prand > skew-data.dat
```

If you run your `skew.cpp` program using this data set, it should print something like this:

```
N is 100000
Mean is 2.998120
Stddev is 1.731314
Skew is 0.580567
Compare to 0.012247
Ratio is 47.403123
```

The ratio at the end is g/σ_{ns}, which is much larger than 1, showing that this set of data has a significant skew.

Some Programs for Generating Lists of Random Numbers

On the following page you'll see three programs that will generate output that you can use to test the programs you write in response to this chapter's practice problems. The three programs are:

- `urand.cpp` (Program 7.13), which generates 100,000 random numbers uniformly distributed between zero and one. This is just a flat distribution, where any number in that range is equally likely.
- `nrand.cpp` (Program 7.14), which generates 100,000 random numbers in a *normal* or *gaussian* distribution centered at zero. These are numbers that are most likely to be near zero, but sometimes have higher or lower values, with decreasing probability as the number gets farther from zero.
- `prand.cpp` (Program 7.15), which generates 100,000 random numbers in a *Poisson* distribution. In this case, the numbers are all greater than or equal to zero, and are most likely to be equal to 3, but might be smaller or larger than 3. Smaller numbers are less likely as they approach zero, and larger numbers are less likely as they get farther from 3. This distribution isn't symmetric about the mean.

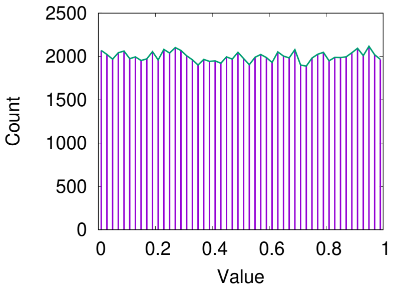


Figure 7.54: A histogram of the random numbers created by `urand.cpp`.

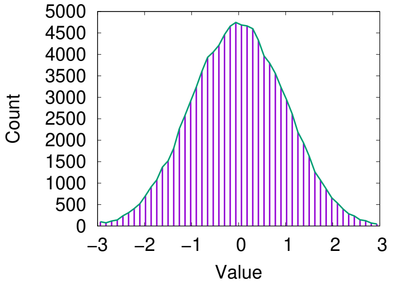


Figure 7.55: A histogram of the random numbers created by `nrand.cpp`.

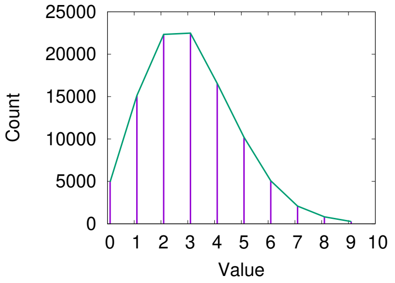


Figure 7.56: A histogram of the random numbers created by `prand.cpp`.

Program 7.13: urand.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
  int i;
  srand(time(NULL));
  for ( i=0; i<100000; i++ ) {
    printf( "%lf\n", rand()/(1.0+RAND_MAX) );
  }
}
```

Program 7.14: nrand.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
int main () {
  int i;
  int iroll;
  int nroll = 12;
  double sum;
  srand(time(NULL));
  for ( i=0; i<100000; i++ ) {
    sum = 0;
    for ( iroll=0; iroll<nroll; iroll++ ) {
      sum += rand()/(1.0+RAND_MAX);
    }
    printf ( "%lf\n", sum-6.0 );
  }
}
```

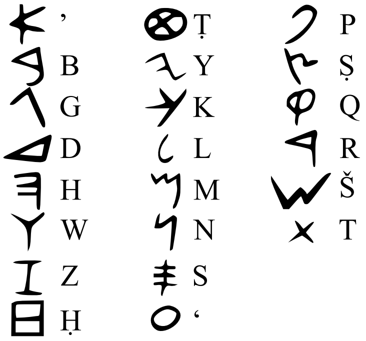
Program 7.15: prand.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
int main () {
  int i;
  int x;
  double u, p, s;
  double mu = 3;
  srand(time(NULL));
  for ( i=0; i<100000; i++ ) {
    x = 0;
    p = exp( -mu );
    s = p;
    u = rand()/(1.0+RAND_MAX);
    while ( u > s ) {
      x++;
      p = p*mu/x;
      s += p;
    }
    printf ( "%d\n", x );
  }
}
```

8. *Character Strings*

Some very early text: The Epic of Gilgamesh, first written down around 2,000 BCE. It tells the story of King Gilgamesh and his friend Enkidu and their epic journey to visit the wise man Utnapishtim, who was a survivor of the Deluge. New fragments of the Epic were discovered in an Iraqi museum in 2015.

Source: Wikimedia Commons



The Phoenician alphabet.

Source: Wikimedia Commons

8.1. Introduction

Until now we’ve avoided reading or writing text in our programs, and have worked exclusively with numbers. Even though we use a lot of numbers in science and engineering, we still need to work with words sometimes. Wouldn’t it be convenient to have a header at the top of each column of numbers in a data file, saying what that column means? We might also want to use text for data values themselves, like “`on`” or “`off`” instead of zero or one, to make our data easier for humans to read. Even if we have a glossary of numerical values and their meanings, like “32 = Virginia, 33 = Maryland”, it’s handy to be able to just look at the data in a file and see its meaning directly, without having to go look up the meaning of each number.

Early writing systems used written symbols to represent the sounds of speech. Learning to read requires that you learn a sort of glossary of these symbols and their speech equivalents. Computers can only store data in the form of binary numbers, so somewhere there’s going to need to be a glossary that matches up text with numerical equivalents.

In this chapter we’re going to see how computers store text, and how to read, write and compare text in a C program. Although you might not expect it, introducing text also introduces a lot of potential problems for the programmer.

8.2. Character Variables

As we’ve seen, there are several different types of variables in C. We’ve used “`int`” for integers and “`double`” for floating-point numbers. Now we’re going to introduce another type of variable: “`char`”. A `char` variable can hold one character (letter, number, punctuation, etc.)

Here's a C statement that defines a `char` variable named `letter` and gives it the initial value `'A'`:

```
char letter = 'A';
```

Notice that we use single-quotes (apostrophes) around the letter. This tells the computer that `A` isn't the name of a variable, it's literally just the letter `A`. Program 8.1 shows how you might use `char` variables.

Program 8.1: checkyn.cpp

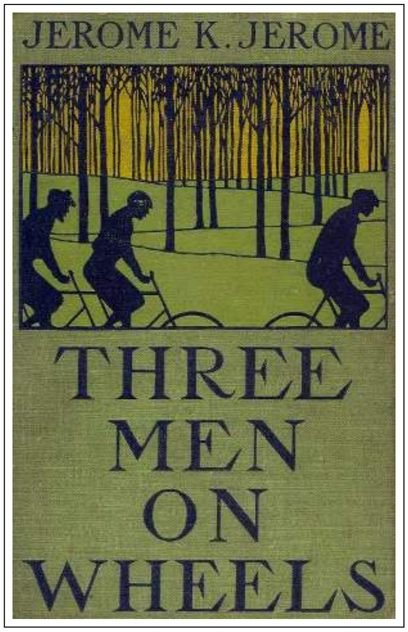
```
#include <stdio.h>
int main () {
  char answer;

  printf ("Can you ride a bike? (y or n): " );
  scanf ("%c", &answer);

  if ( answer == 'y' ) {
    printf ("Yay! Biking is fun.\n");
  } else if ( answer == 'n' ) {
    printf ("Awww. You should learn.\n");
  } else {
    printf ("Might ride a bike, but can't follow instructions.\n");
  }
}
```

Along with the new variable type, we need a new type of placeholder for our printf and scanf statement. Just as we use "`%d`" for `int` and "`%lf`" for `double`, we use "`%c`" for `char`. When we say "`%c`" we mean "insert a single character here".

8.3. Character Strings

We can use an array of `char` elements to hold a chunk of text. We call such an array a "character string" (see Figure 8.1). We'll use the terms "character array", "character string", and "string" interchangeably.

As we saw in Chapter 6, C lets us put numbers into an array when we define it (although this is only practical for small arrays). For example, we could define a small array of integers and print them out like this:



Three Men on Wheels (1900, aka *Three Men on the Bummel*) by Jerome K. Jerome is a sequel to *Three Men in a Boat (to Say Nothing of the Dog)* (1889). It follows Jerome, George, and Carl on a bicycle trip through Germany.

```
int array[5] = {1,2,3,4,5};
int i;
for ( i=0; i<5; i++ ) {
  printf ( "%d\n", array[i] );
}
```

We could do something similar with an array of characters if we wanted to:

```
char string[20] = {'t','h','i','s',' ',
                   'i','s',' ','a',' ',
                   't','e','s','t','.'};
int i;
for ( i=0; i<20; i++ ) {
  printf ( "%c", string[i] );
}
```

Since we've omitted the `\n` all of the characters will be printed on the same line, and the output will say "`this is a test.`" That's a really tedious way to define a chunk of text and print it out. Fortunately, C provides with a couple of shortcuts to make it easier.

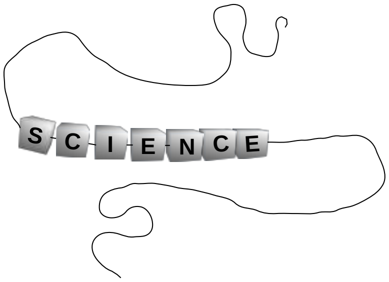


Figure 8.1: Think of a character string as being like a string of letter beads.

First of all, there's a special way of setting the inital value of character strings. Instead of using curly brackets and a list of single-quoted characters, we can just enclose the text in double-quotes:

```
char motto[10] = "Science!";
```

Second, there's a special placeholder, "`%s`", for printing character strings all at once, instead of one character at a time:

```
printf ( "%s\n", motto );
```

Notice that we don't have to use all of the elements of a character array. In the example above, the text "Science!" is only eight characters long, but we've defined `motto` to have ten elements. In fact, if we don't plan on ever putting more text into a character string, we can ask the compiler to figure out its length automatically, by just leaving the length blank:

```
char motto[] = "Science!";
```

Of Course, we'll run into trouble if we try to stuff more characters into a character array than it will hold. This would create the same problems we saw in Chapter 6 with other kinds of arrays.

In the following we're going to look at several tiny programs that illustrate some of the problems you might run into when you use character strings in your programs. In each case, we'll show you the "right" way to do it

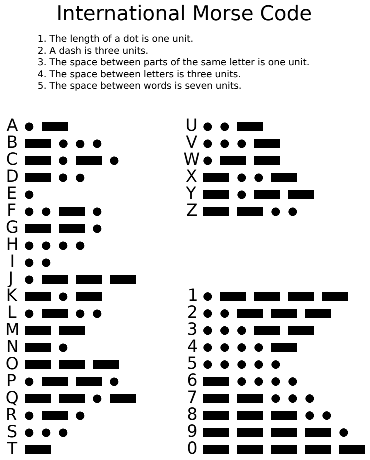

International Morse Code
1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.

Figure 8.2: Morse Code replaces letters with patterns of dots and dashes.

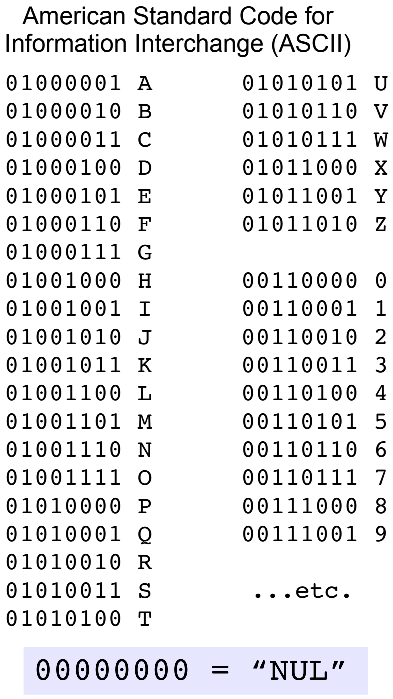
Source: Wikimedia Commons

8.4. How Strings Are Stored

Prior to the 1960s, the most widespread way of communicating data electronically was morse code (see Figure 8.2). When a telegram was sent, its text was encoded in morse code and transmitted through air or a wire to its destination, where it was decoded back into text.

Morse code was fine for human telegraphers, but it was clumsy for computers. In the 1960s the "American Standards Association" published a new, more computer-friendly way of transmitting text. This was called the American Standard Code for Information Interchange (ASCII).

In ASCII, each character is represented by 8 bits of information (1 byte). When you store text in a file on disk, the text is stored as ASCII characters. (Actually, other encodings like UTF-8 may be used these days because they allow multi-national characters, but the principle is the same. For simplicity, let's just assume everything is ASCII.)



American Standard Code for Information Interchange (ASCII)

01000001	A	01010101	U
01000010	B	01010110	V
01000011	C	01010111	W
01000100	D	01011000	X
01000101	E	01011001	Y
01000110	F	01011010	Z
01000111	G		
01001000	H	00110000	0
01001001	I	00110001	1
01001010	J	00110010	2
01001011	K	00110011	3
01001100	L	00110100	4
01001101	M	00110101	5
01001110	N	00110110	6
01001111	O	00110111	7
01010000	P	00111000	8
01010001	Q	00111001	9
01010010	R		
01010011	S	...etc.	
01010100	T		

00000000 = "NUL"

Figure 8.3: ASCII code replaces letters with zeros and ones.

8.5. The Length of Strings

Take a look at Figure 8.4, where we define a 10-element character array called `name` and put the word "Fred" into it. If we wanted to print the text stored in `name` we might write a C statement like this:

```
printf ( "%s\n", name );
```

That looks straightforward enough, but it leads to a puzzle: The character array `name` has ten elements, but we're only using four of them. How does the `printf` function know when it gets to the end of the text? In fact, as we noted in Chapter 6, C doesn't prevent us from reading or writing past the end of an array. Shouldn't we have to tell printf how many characters are in our text, or at least tell it how many elements are in the `name` array? With other types of array, we haven't been able to just say "print the array", so why are we able to do so with character arrays?

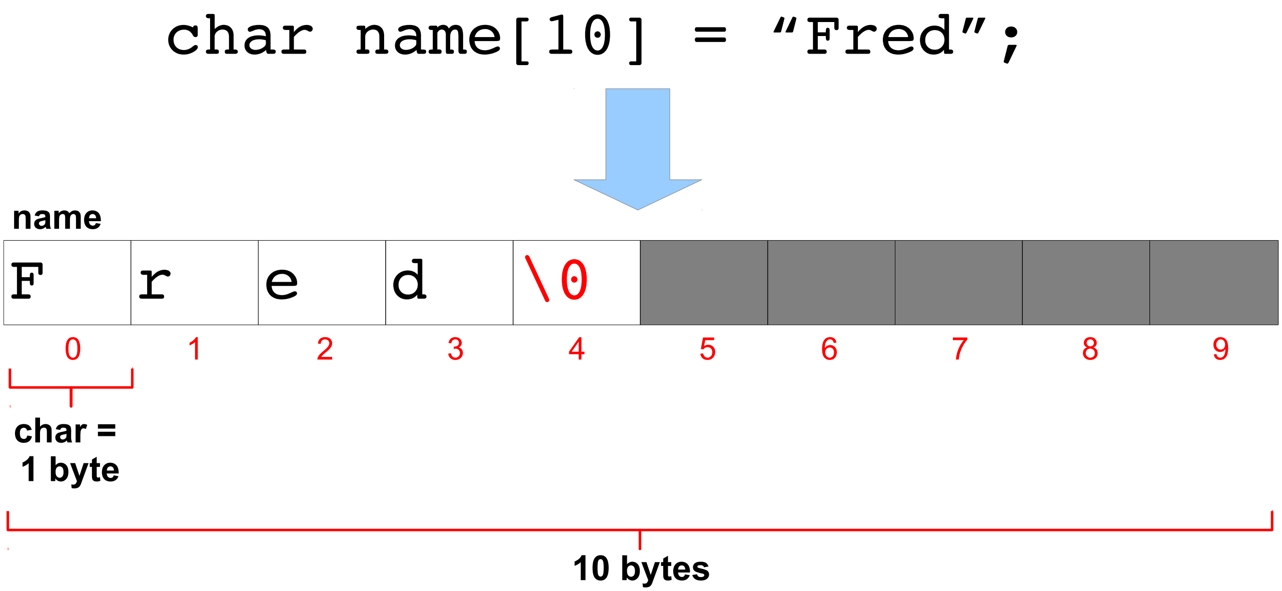


Figure 8.4: The end of a string is indicated by a special non-printable character, the "NUL" character, which we represent by '\0' here (see Figure 8.3). Its ASCII representation is "`00000000`".

The answer is that the end of the text in a character array is marked by a special ASCII character, the "NUL" character, which has the ASCII code `00000000`. When we define a character string as in Figure 8.4 we need to be sure to leave room for the longest text it will ever contain *plus one extra element* to hold the trailing NUL character.

Without the NUL character, `printf` would just keep on printing bytes until it happened to find a NUL somewhere in memory or caused the program to crash, since it wouldn't know where the character array ended. In C programs, we represent the NUL character by `\0`.

Each character of a string is stored in memory as ones and zeros, according to the ASCII code. Figure 8.5 shows an example of what you might find in memory if your program contained the statement "`char day[] = "Tuesday";`"

Whenever you use *nano* to create a text file (one of the cpp files you've been writing, for example), the things you type are stored as ASCII-encoded characters in a file on the computer's disk. If you could see the actual bits, and you understood ASCII, you could read the file's contents.

	day
T	01010100
u	01110101
e	01100101
s	01110011
d	01100100
a	01100001
y	01111001
\0	00000000

Figure 8.5: This is how the word "Tuesday" would be represented as ones and zeros in memory, stored in an eight-element character array named `day`.

8.6. The `strlen` Function

Can we get our program to tell us the length of a character string? Sure thing! We can use the `strlen` function for this. For example:

```
char name[20] = "Bryan";
int length;
length = strlen(name);
printf ( "This name is %d characters long.\n", length );
```

Some versions of the C compiler might give you an error message if you try to use `strlen` directly as an argument to a function like `printf` or in comparison with an integer in an "`if`" statement. That's because `strlen` doesn't really return an `int` value. Instead of an `int`, `strlen` uses a special data type named `size_t`.

If we tried to write a program containing a statement like this:

```
printf ( "This name is %d characters long.\n", strlen(name) );
```

the C compiler would complain that we've told `printf` to expect an `int` (by using a `%d`), but strlen returns a `size_t`. The complaint would look something like this:

```
program.cpp:6:62: warning: format '%d' expects argument of type 'int',
but argument 2 has type 'size_t {aka long unsigned int}' [-Wformat=]
  printf ( "This name is %d characters long.\n", strlen(name) );
```

The cure for this is to either use a variable like `length`, as we did in the first example above, or to explicitly tell the C compiler to convert `strlen`'s value into an `int`. We could do that like this:

```
printf ( "This name is %d characters long.\n", (int)strlen(name) );
```

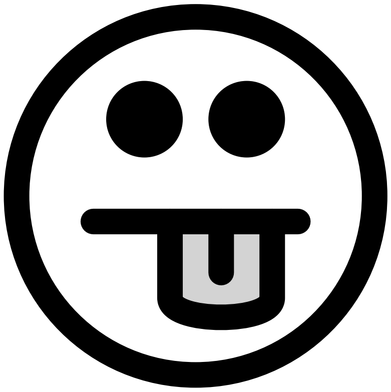
We've talked about this kind of re-casting of values in Chapter 2 and Chapter 3.

8.7. Comparing Strings

Imagine that we have two character strings, and we want to compare them to see if they're the same. We might try something like the following:

Program 8.2: scomp.cpp (Why doesn't this work?)

```
#include <stdio.h>
int main () {

  char s[] = "junk";
  char t[] = "junk";

  if ( s == t ) {
        printf ("They match.\n");
  } else {
        printf ("They don't match.\n");
  }
}
```



Why doesn't this work? Because "`s`" and "`t`" are arrays. Think about it: if we had two `int` variables, `x` and `y`, we could compare their values with "`if (x==y)`". Similarly, if we had two arrays of `int` elements , `a[10]` and `b[10]`, we could compare two of their elements with "`if (a[1] == b[1])`". But what would we mean if we typed "`if(a==b)`"?

It turns out that, in C, if you type just the name of an array, you get the memory address of the beginning of the array[1]. Since "`s`" and "`t`" in the example above are two different arrays, each of which has its own allocated section of memory, each of them will have a different address. So, "`if(s==t)`" will never be true.

If you compile and run Program 8.2 you'll see that it always says "They don't match." This is obviously not the right way to compare two strings.

One way to solve the problem would be to write a "`for`" loop and compare each character in the two strings, one by one. This would be inconvenient though, especially if we had to do it often. Fortunately, C provides us with a function that can compare strings for us. It's called "`strcmp`" (for "string compare").

If we have two character strings, `s` and `t`, and give them to `strcmp`

[1] We'll learn more about this later.

like this:

```
result = strcmp( s, t );
```

the value of the result will tell us whether the two strings are the same. There are three possiblities:

result = 0	The two strings are identical.
result > 0	s is "greater" than t
result < 0	s is "less" than t

In this context "greater than" and "less than" refer to the dictonary order of the two strings. If s would come before t in a dictionary, strcmp says that s is less than t. According to strcmp, "aardvark" is less than "zebra".

Program 8.3 shows the right way to compare two strings.

Program 8.3: scomp.cpp (Doing it the right way.)

```
#include <stdio.h>
#include <string.h>
int main () {
  char s[] = "junk";
  char t[] = "junk";
  if ( strcmp( s, t ) == 0 ) {
        printf ("They match.\n");
  } else {
        printf ("They don't match.\n");
  }
}
```

Notice that we need to add a new #include line before we can use the strcmp function.

Instead of saying "strcmp(s,t) == 0" in our "if" statement, we could have saved some typing by saying "!strcmp(s,t)". When we say "if (CONDITION)", the CONDITION is true if it has a non-zero value, and false otherwise. Because strcmp returns 0 if the strings are equal, we need to use a ! (read "not") to logically invert this into a true value. You might read such an "if" statement as "if strcmp *doesn't* return a *non*-zero value...".

Exercise 41: Comparing Strings

Create, compile, and run Program 8.3. Does it do the right thing?

Try changing one of the strings, recompiling, and running again. Does the program properly tell you that the two strings are different now?

8.8. Reading Strings

We've used `scanf` and `fscanf` to read numbers. Now we'd like to use these functions to read text. Can we do it?

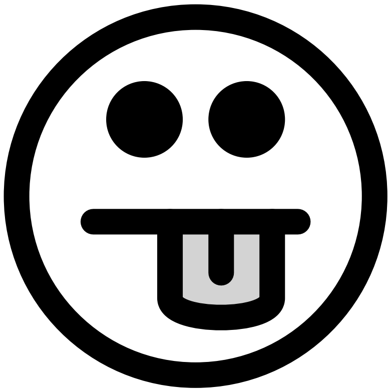
There are some complications, and to understand them we'll need to know a little more about how `scanf` and `fscanf` work. Until now, we've taken it on faith that we needed to put an ampersand (`&`) in front of variable names when reading numbers with these functions. The reason that's true is because `scanf` and `fscanf` want the *memory address* of a variable.[2] If I have a variable named `height`, then "`&height`" will be the address of the chunk of memory that the computer has assigned to that variable.

As we saw in our bad string comparison example, the name of an array is actually just the memory address of the beginning of the array. This means we can leave off the "`&`" when we read a character array with scanf.

There are still other complications, though, which we can illustrate with Program 8.4. This program asks you to enter some text, and then just tells you what you entered.

Program 8.4: sread.cpp (Not quite getting it right.)

```
#include <stdio.h>
int main () {
  char string[10];
  printf ("Enter some text: ");
  scanf( "%s", string );
  printf( "You said %s\n", string );
}
```



The program defines a character array named `string`, and then uses

[2] We'll learn why this is so when we study functions in Chapter 9.

scanf with the %s format specifier to read some text into this array. Notice that the program omits the ampersand we'd use in scanf if we were reading a number.

If you try giving this program a word like "Hello", it seems to work fine. In fact, any short, single word will work. But what if we give it something longer, like "abcdefghijklmnopqrstuvwxyz"? Then you'll find that the program crashes with a "Segmentation Fault" error. That's because we've tried to go past the end of the string character array, which only has room for ten characters. This is the same kind of problem we had with numerical arrays in Chapter 6.

We can fix our program by just adding one letter: change "%s" to "\%**9**s" in the scanf statement. This tells scanf to read *no more than nine characters*. Why nine instead of ten? Because we need to leave room for a NUL character at the end, to mark the end of the string. Now, if we type "abcdefghijklmnoprstuvwxyz" the program will print "abcdefghi" (just the first nine characters of the text we entered).

Program 8.5: sread.cpp (OK for some things.)

```
#include <stdio.h>
int main () {
  char string[10];
  printf ("Enter some text: ");
  scanf( "%9s", string );
  printf( "You said %s\n", string );
}
```

Note that everything we've said about scanf applies to fscanf as well.

Exercise 42: Safe String Reading

Create, compile and run Program 8.5. Try giving the program some words without spaces, and then try giving it sentences with spaces in them. Does it behave as expected? What if you type a tab character instead of a space?

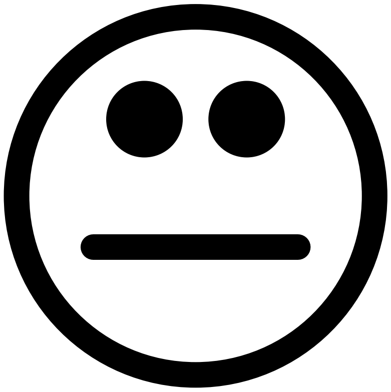
There's still one problem left, though. Even the improved version of the program has trouble when we enter text with spaces in it. If we enter "this is a test", the program says we typed "this".

That happens because scanf stops reading text (%s) when it sees a "white space" character (a space or a tab). This may not be what you want your program to do. If you need to read strings containing spaces, a better choice is "fgets". The fgets function reads a specified number of characters from a file. Even though we're reading from the keyboard, not a file, we can still use fgets.

Remember that we saw in Chapter 7 that three "files" are automatically opened whenever we run a program: stdout, stderr, and stdin. The first two usually point to your display, and the third (stdin) usually points to your keyboard. We can use fgets in our program by telling it to read from stdin. That's what Program 8.6 does.

Program 8.6: sread.cpp (Better, but see next section...)

```
#include <stdio.h>
int main () {
  char string[10];
  printf ("Enter some text: ");
  fgets( string, 10, stdin );
  printf("You said %s\n",string);
}
```



See next section for caveats.

The fgets function takes three arguments: The name of a character string variable in which to store what we read, the size of that character string, and a file handle pointing to an open file to read things from. fgets will read, at most, one less than the size of the character string, automatically leaving space for the trailing NUL character.

But what about...?

So why does "%s" stop at white spaces? It's so we can do things like this:

```
char name[10];
int year;
printf ( "Enter your last name and birth year: ");
scanf("%9s %d", name, &year);
```

or like this:

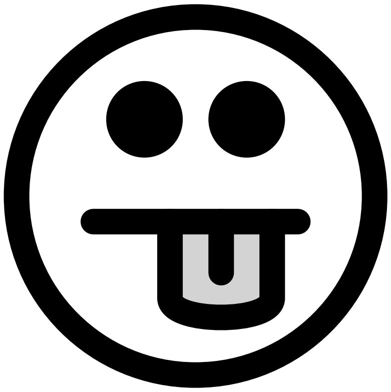
```
char firstname[10], lastname[10];
scanf("%9s %9s", firstname, lastname);
```

If scanf didn't stop at white spaces, the first example would try to stick things like "Wright 1961" into "name". It would never know when you were done typing the first word, and had started typing something else.

If you want the things you enter to be broken up into words, `scanf` is a good choice. If you want everything to be put into one variable, `fgets` is the thing to use.

8.9. Line Endings

There's still a potential problem with Program 8.6 though, and it's a subtle one. To illustrate it, let's make a small change to the program and try running it again. The new version is Program 8.7.



Program 8.7: sread.cpp (Watch what happens now...)

```
#include <stdio.h>
int main () {
  char string[10];
  printf ("Enter some text: ");
  fgets( string, 10, stdin );
  printf("You said %s. You really did.\n",string);
}
```

If we ran Program 8.7 and entered the text `hello`, we'd see something like the following:

```
Enter some text: hello
You said hello
. You really did.
```

What's going on here? Shouldn't the program have written "You said hello. You really did.", all on one line? The difference is due to the fact that `fgets` interprets the "enter" key as an ASCII "newline" character, and it puts that newline into `string` just like the other characters you typed. In some circumstances that might be OK, but we'll often want to get rid of the extra newline.

Dealing with line endings can be especially tricky if your program reads text from a file. For historical reasons, each of the three most popular operating systems (Windows, OS X, and Linux) uses a different way of indicating the end of a line in an ASCII file. OS X, for example, uses the ASCII "CR" ("Carriage Return") character, which we can write as "`\r`" in C programs. Linux, on the other hand, uses the ASCII "LF"

("Line Feed") character, which we can write as "\n". Windows uses *both*, putting "\r\n" at the end of each line.

To make our programs as portable as possible, it would be nice if they could deal with any of these.

To eliminate such spurious characters we first have to find them. Let's start by looking at a handy C function for finding particular characters in a string. Consider Program 8.8.

Program 8.8: findchar.cpp

```
#include <stdio.h>
#include <string.h>
int main () {
  char welcome[] = "Testing, testing. Are you there?";
  int i;

  i = strcspn( welcome, ".,?" );

  printf("The first punctuation is character number %d\n", i);
}
```

The strcspn function has a name that's hard to remember[3], but what it does is simple. You give strcspn a string and a list of characters you're interested in, then it steps through the string, one character at a time, until it finds an interesting one. When it finds the first interesting character it tells you its location.

Program 8.8 defines a character string named welcome. The program uses strcspn to find the location of the first punctuation character in this string. Remember that a character string is just an array of char variables, and that array indices begin with zero. If you start with zero and count characters, you'll find that the "," (the first punctuation mark) is element number 7 of welcome, and that's what Program 8.8 would tell you if you compiled and ran it.

In principle, we could use the strcspn to find \r and \n characters. Once we've found them, we need to know how to get rid of them. That turns out to be easy.

Remember again that a character string is just an array of characters.

[3] It's an abbreviation for "string complementary span", but that's no more memorable.

Once we know which array element holds a letter we want to change, all we need to do is put a different character into that element.

Let's get back to the most recent version of our `sread` program now (Program 8.7). Take a look at Figure 8.6. At the top we see the contents of `string` as Program 8.7 would see it right after the user types "hello" and presses the enter key.

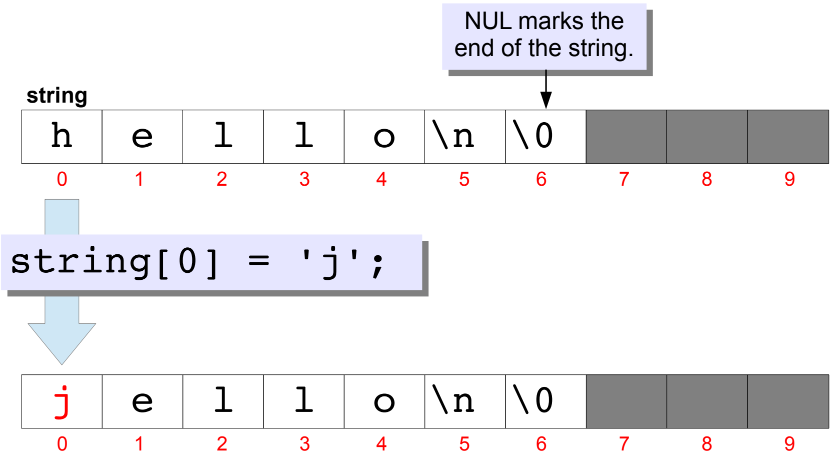


Figure 8.6: Changing one character in a string.

`string` is a 10-element character array. The next-to-last character is "newline", which we represent in C programs as `\n`. Following the newline is an ASCII NUL character, represented by `\0`, which marks the end of the string, as described in Section 8.5 above.

If we wanted to change "hello" into "jello", we could say:

```
string[0] = 'j';
```

making the first letter (element number zero) of `string` a "j" instead of an "h", as shown at the bottom of Figure 8.6.

Now take a look at Figure 8.7. If we wanted to get rid of the newline in `string`, we could replace character number 5. But what should we replace it with? What if we put in another `\0`, as in the bottom of Figure 8.7? Now the newline is gone, and the newly inserted `\0` marks the new end of the string. (The second `\0` is ignored.) We've chopped the troublesome newline off the end of the string!

So, our two-part strategy for removing trailing `\r` and `\n` characters is (1) use `strcspn` to locate them and (2) write an ASCII NUL character in their place. Program 8.9 shows a final version of our `sread` program

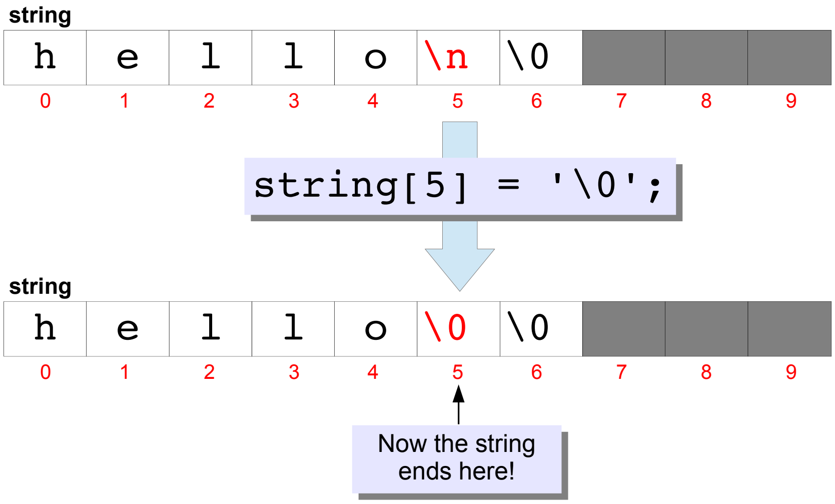


Figure 8.7: Replacing a newline with a NUL.

that implements this strategy. As you can see, we only need to add two lines to the program.

Exercise 43: Space, The Final Frontier

Now modify Program 8.5 so that it looks like Program 8.9. Try it again with input that includes spaces or tabs. How does it behave differently?

Program 8.9: sread.cpp (Now deals with spaces and line endings)

```
#include <stdio.h>
#include <string.h>
int main () {
  char string[10];
  printf ("Enter some text: ");
  fgets(string,10,stdin);
  string[ strcspn( string, "\r\n" ) ] = '\0';
  printf("You said %s. You really did.\n",string);
}
```

If we ran Program 8.9 and typed “hello”, the result would look like this, as the user would expect:

```
Enter some text: hello
You said hello. You really did.
```

The `strcspn` function gives the location of the first `\r` or `\n`, then the program puts a `\0` at that spot. This would be safe even if the string didn't contain any `\r` or `\n` characters. In that case, `strcspn` returns the location of the `\0` that's already at the end of the string, and the program wouldn't end up changing anything.

It's generally a good idea to use `strcspn` in this way to trim off any extra `\r` or `\n` characters. I recommend you do this whenever you use `fgets`.

Note that in Program 8.9 we could have done things in two explicit steps, by defining an integer variable `i` and saying:

```
i = strcspn( string, "\r\n" );
string[i] = '\0';
```

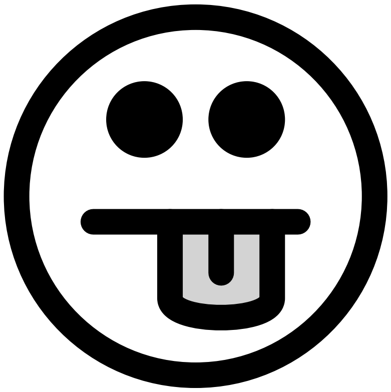
Either way is fine. Feel free to do it this way if you find it easier to understand.

8.10. Assigning Values to Strings

Since strings are arrays, we also need to take care when assigning values to them in our programs. Take a look at Program 8.10 for example. This looks pretty straightforward. We have two character string variables, `s` and `t`, and we want to set `t` equal to `s`, just like we've been doing with numerical variables.

Program 8.10: sassign.cpp (This won't work)

```
#include <stdio.h>
int main () {
  char s[10] = "Testing";
  char t[10];

  t = s;
  printf( "%s\n", t);
}
```



You'll find that g++ refuses to compile this program though. If you try, you'll probably see an error message like this:

```
sassign.cpp: In function 'int main()':
sassign.cpp:6: error: invalid array assignment
```

Why does this happen? Remember that `t` and `s` are arrays, not single values. The C compiler is telling you that it can't figure out what you want to do here.

What we'd like to do is make each element of the `t` array be the same as the corresponding element of the `s` array. We could write a "`for`" loop to go through all of the array elements and do that, but there's an easier way to do it with character arrays.

We can use the "`spnrintf`" function to "print" the value of one string into another string. This is what we do in Program 8.11.

Program 8.11: sassign.cpp (The right way.)

```
#include <stdio.h>
int main () {
  char s[10] = "Testing";
  char t[10];

  snprintf( t, 10, "%s", s);
  printf( "%s\n", t);
}
```

The snprintf function is like printf, but it takes two extra arguments: the name of a string, and the number of characters. In Program 8.11 snprintf will write a maximum of ten characters into the character string named t. It's important that snprintf lets us specify the maximum number of characters, so we don't write past the end of t.

We could also to things like this:

```
snprintf (t, 10, "Hello world!\n");
```

which would put the text "Hello world!" into t.

Internally, snprintf just does the same thing as looping through all of the characters in the arrays, one by one, and setting their values.

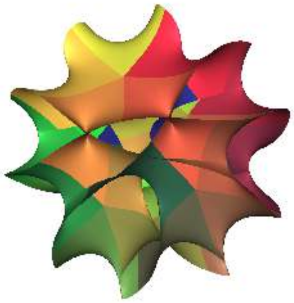
> ### Exercise 44: For Internal Use Only
>
> Create, compile and run Program 8.11. Try modifying the program by replacing "Testing" with something longer that includes spaces. (You may need to increase the size of the s and t character arrays.) Recompile the program and make sure it does what you expect.

8.11. Summary of Good String Usage

In the preceding sections we've gone through a bunch of best practices for using character strings. Let's summarize what we've learned:

Comparing Strings

We can't compare strings the same way we compare numbers. If we try to do so, we'll always be misled into thinking that the strings are different, even if they're not. To do it right, use strcmp to compare



Unfortunately, String Theory has nothing to do with character strings, but this Calabi-Yau manifold is too attractive to leave out.

Source: *Wikimedia Commons*

strings. (You'll need to add `#include <string.h>` to use `strcmp`.) Remember that `strcmp` returns zero if the strings are equal. Here's a usage example:

```
if ( strcmp( s, t ) == 0 ) {
    printf( "They're the same!\n" );
}
```

Reading Strings from the User

C provides us with a special format placeholder, `%s`, for reading strings. Since a character string is an array, we need to take care not to go past the end of the array. There are two good ways to read strings: one for when you want each "word" (separated by spaces) to go into its own variable, and another for when you want everything the user types (including spaces) to go into a single variable.

- If you want to split the input wherever there's a space, use `scanf`. Always specify the number of characters by putting a number between `%` and `s`. The number should be one less than the length of the character string array, to leave room for a NUL character at the end. Here's a usage example, suitable for reading text into a 10-character-long string:

  ```
  scanf ( "%9s", string );
  ```

- If you want to put all of the input, spaces and everything, into one variable, use the `fgets` function. Be sure that the size you give it matches the actual size of the character string variable. `fgets` will automatically leave room for the trailing NUL character. Also, use `strcspn` to trim off trailing newlines. Here's a usage example suitable for a 10-character long string:

  ```
  fgets ( string, 10, stdin );
  string[ strcspn( string, "\r\n" ) ] = '\0';
  ```

Assigning Values to Strings

We can't just assign values to character string variables the same way we do with numerical variables. Instead, use snprintf to "print" text into the variable. Here's a usage example that would copy the contents of the variable `s` into the 10-character-long variable `t`:

```
snprintf( t, 10, "%s", s);
```

Writing past the end of a string array is a very common programming bug. It often leads to crashes, and is responsible for many security flaws. Sticking to the methods above will help you avoid these problems in your programs.

8.12. Reading a Gradebook

Let's look at a practical program that uses the string techniques we've been talking about. In this example we'll be reading students' names and grades from a gradebook file and calculating grade averages.

Albert Gleizes, *Composition pour Jazz* (1915)

Source: Wikimedia Commons

Take a look a Program 8.12. It reads names and columns of grades from a file like this:

```
Davis           9.2   9.8   9.8   10.   9.2   9.1
Gillespie       8.7   8.7   8.7   8.6   8.9   9.2
Monk            10.   9.0   9.5   9.0   9.1   9.8
Vaughan         9.9   9.9   9.8   8.5   9.0   9.8
Coltrane        9.0   9.1   8.9   9.9   9.7   8.6
Mingus          8.9   9.8   8.6   9.8   9.9   9.8
Parker          9.6   10.   9.1   9.1   9.8   8.9
Holliday        9.2   8.7   10.   8.9   9.8   9.0
Armstrong       8.6   8.6   9.0   9.2   8.6   8.7
Ellington       9.8   9.6   9.6   9.6   10.   10.
Fitzgerald      9.8   9.2   9.9   9.8   8.7   9.6
```

The first column is the student's last name, and the other columns are grades for each of six homework assignments.

Program 8.12 uses `fscanf` to read the students name and store it in the 20-character-long string variable named `lastname`. To make sure it doesn't go past the end of `lastname`, the program tells `fscanf` to use the format "`%19s`", limiting the number of characters to 19 at most, and leaving at least one space to store the terminating NUL character marking the end of the string.

This program uses a technique similar to the one used in our census program in Chapter 7 for reading the multi-column data in the file `grades.dat`. A "`for`" loop reads `ngrades` numbers from each line of the file. Unlike the census program, we don't read the numbers into an array, since this program doesn't care which number was in which column. We only want to add them up, so we can calculate the mean.

The last line of Program 8.12 prints out each student's name and mean grade. Notice that we tell `printf` to print only the first two decimal

Program 8.12: grades.cpp

```
#include <stdio.h>
int main () {
  int ngrades=6;
  char lastname[20];
  double sum, grade;
  int i;
  FILE *gradebook;

  gradebook = fopen("grades.dat","r");
   while ( fscanf( gradebook, "%19s", lastname ) == 1 ) {
               sum = 0.0;
               for ( i=0; i<ngrades; i++ ) {
                fscanf(gradebook, "%lf", &grade);
                sum += grade;
               }
               printf ( "%s %.2lf\n", lastname, sum/ngrades );
   }
}
```

Read all lines from file

Get all grades from each line

places of the numbers by using "`%.2lf`". As we saw in Chapter 3, a format like "`%`**`n.m`**`lf`" means "show m characters with **n** to the right of the decimal place." (We can omit the **n** if we just want to specify the number of decimal places.)

If we ran Program 8.12 we'd see something like this:

```
Davis 9.52
Gillespie 8.80
Monk 9.40
Vaughan 9.48
Coltrane 9.20
Mingus 9.47
Parker 9.42
Holliday 9.27
Armstrong 8.78
Ellington 9.77
Fitzgerald 9.50
```

The program seems to be doing its job, but the output could be more readable. It would be nice if things lined up in straight columns. If we change the last `printf` statement we could make things a little prettier:

```
printf ( "%20s %.2lf\n", lastname, sum/ngrades );
```

We've changed `%s` into `%20s`. If we ran the modified program, the result would look like this:

```
               Davis 9.52
           Gillespie 8.80
                Monk 9.40
             Vaughan 9.48
            Coltrane 9.20
              Mingus 9.47
              Parker 9.42
            Holliday 9.27
           Armstrong 8.78
           Ellington 9.77
          Fitzgerald 9.50
```

What happened? When we say `%20s` we mean "make the output string exactly 20 characters long, padding it on the front with spaces if there's not enough text to fill the full 20 characters."

If we don't like this right-justified style, we can move the text over to the left by changing `%20s` into `%-20s`:

```
Davis                9.52
Gillespie            8.80
Monk                 9.40
Vaughan              9.48
Coltrane             9.20
Mingus               9.47
Parker               9.42
Holliday             9.27
Armstrong            8.78
Ellington            9.77
Fitzgerald           9.50
```

Exercise 45: Reading and Writing Text

Here's a challenge for you. Write a program named `classes.cpp` that asks the user how many classes he or she has on each day of the week. After collecting the data, the program should write the name of each weekday and the number of classes on that day into a data file named `classes.dat`

The program should have a loop that asks the user to enter the name of the day of the week and the number of classes on that day. If the user enters "quit" as the day, the loop should stop.

The program should start out something like this:

```
#include <stdio.h>
#include <string.h>
int main () {
  char day[10];
  int classes;
  FILE *output;
  output = fopen( "classes.dat", "w" );
```

It would be a good idea to use two separate `scanf` statements to read the day name and the number of classes, instead of trying to read both with the same `scanf`. (Can you think of a reason why this is so?)

Here are some hints:

- Remember that you don't need a `&` in front of the variable name when you read a character string with `scanf` (but you do when you read a number).
- You can test to see if a `day` contains the text "quit" like this:

  ```
  if ( strcmp( day, "quit" ) == 0 )
  ```

- You can write things into a file using `fprintf`, like this:

  ```
  fprintf ( output, "%s %d\n", day, classes );
  ```

Compile and run your program. The file it creates (`classes.dat`) should look like this:

```
Monday 4
Tuesday 2
Wednesday 3
Thursday 2
Friday 3
```

This is similar to the data files we've graphed with *gnuplot* in the past, except that one of the columns contains text. Start up *gnuplot* and type the following to cause it to use the days of the week as labels on the X axis:

```
set xrange [-1.5:5.5]
set yrange [0:6]
plot "classes.dat" using 2:xticlabels(1) with boxes
```

The first two commands set the range of the X and Y axes so that the data will fit nicely on the graph. The third command tells *gnuplot* to plot the second column of the data, and use the first colum as the labels on the X axis. The result should look something like Figure 8.8.

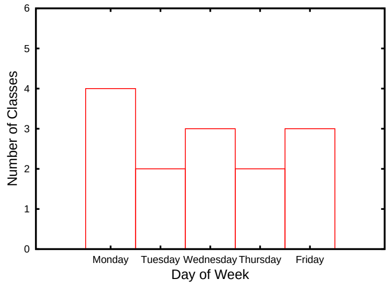


Figure 8.8: Your class schedule might look something like this.

Sometimes you might want the X axis labels to be vertical. You can do this by giving *gnuplot* the command "`set xtics rotate by 90`", and then typing "`replot`". Give it a try. What happens if you use `-90` instead of `90`?

8.13. Reading Column Headers

It would be nice if the columns of our gradebook file had headers, telling the name of each column's assignment. Maybe something like this:

```
                HW1    HW2    HW3    HW4    HW5    HW6
Davis           9.2    9.8    9.8    10.    9.2    9.1
Gillespie       8.7    8.7    8.7    8.6    8.9    9.2
Monk            10.    9.0    9.5    9.0    9.1    9.8
Vaughan         9.9    9.9    9.8    8.5    9.0    9.8
Coltrane        9.0    9.1    8.9    9.9    9.7    8.6
Mingus          8.9    9.8    8.6    9.8    9.9    9.8
Parker          9.6    10.    9.1    9.1    9.8    8.9
Holliday        9.2    8.7    10.    8.9    9.8    9.0
Armstrong       8.6    8.6    9.0    9.2    8.6    8.7
Ellington       9.8    9.6    9.6    9.6    10.    10.
Fitzgerald      9.8    9.2    9.9    9.8    8.7    9.6
```

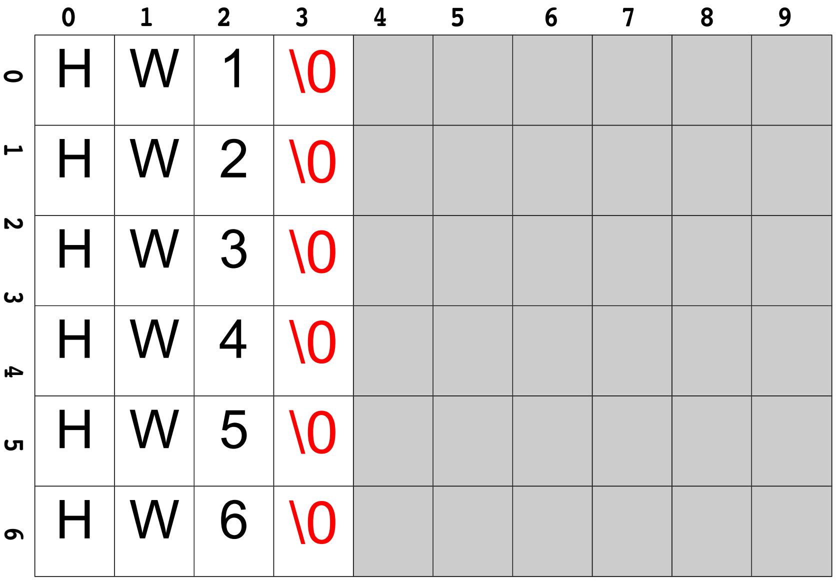
Adi Holzer, *Satchmo (Louis Armstrong)* (2002)

Source: Wikimedia Commons

Program 8.13 on page 293 is designed to read this modified data file. In addition to the things our previous program did, this new program also calculates a class average for each assignment. To do this, it needs to sum up the numbers in each column and divide the sum by the number of students. The sum for each column is stored in an element of the new array named `class_sum`[4]. As the file is read, the students are counted by the variable `nstudents`.

But what about the column headers? Just as we have a class sum for each column, we have a header for each column, and we'd like to save those headers in an array so we can print them out later. But remember that a character string is already an array `char` variables, so we're in need of an *array of arrays*.

That's what the variable named `assignment` is for. It's a six-element array of 10-character strings. Once we've read the column headers into it, we might imagine the array looking like Figure 8.9.



	0	1	2	3	4	5	6	7	8	9
0	H	W	1	\0						
1	H	W	2	\0						
2	H	W	3	\0						
3	H	W	4	\0						
4	H	W	5	\0						
5	H	W	6	\0						

Figure 8.9: An array of character strings holding the column headers from our gradebook file.

Since the column headers are in the first line of the file, they're read first. Program 8.13 uses a "`for`" loop to read the headers into elements of the `assigment` array.

The program then proceeds more or less like Program 8.12, except that the new program also keeps a running sum of each column, in the `class_sum` array, and counts the number of students.

At the end, a new loop goes through all of the assignments, printing out the column header and mean grade for each.

[4] Why do we use `const` when defining `ngrades` here? Look back at page 186 in Chapter 6.

Program 8.13: grades.cpp, Now With Headers!

```
#include <stdio.h>
int main () {
  const int ngrades=6;
  char lastname[20];
  double sum, grade;
  int i;
  double class_sum[ngrades];
  char assignment[ngrades][10];
  int nstudents = 0;
  FILE *gradebook;

  gradebook = fopen("grades-with-headers.dat","r");

  for ( i=0; i<ngrades; i++ ) {
    fscanf( gradebook, "%9s", assignment[i] );
    class_sum[i] = 0.0;
  }

  while ( fscanf( gradebook, "%19s", lastname ) == 1 ) {
    sum = 0.0;
    for ( i=0; i<ngrades; i++ ) {
      fscanf(gradebook, "%lf", &grade);
      sum += grade;
      class_sum[i] += grade;
    }
    printf ( "%-20s %.2lf\n", lastname, sum/ngrades );
    nstudents++;
  }

  printf( "\nClass averages:\n" );
  for ( i=0; i<ngrades; i++ ) {
    printf ( "%10s %.2lf\n", assignment[i], class_sum[i]/nstudents );
  }
}
```

8.14. Handling Errors

Up until now, we've been assuming that the files our programs want to read really exist. But mistakes sometimes happen in the real world. We might accidentally rename or delete a data file, or we might mis-type the file's name when we write it into a program. What happens if a program tries to open a file that doesn't exist? Let's try it and see. Take a look at Program 8.14.

Program 8.14: filecheck.cpp

```
#include <stdio.h>
int main () {
  FILE *input;

  input = fopen( "nosuchfile.dat", "r" );
  // Do some stuff, then close the file...
  fclose ( input );
}
```

If `nosuchfile.dat` doesn't exist, the program will give us an error message saying "`Segmentation fault`"[5]. That's not very helpful, and it might take us a while to figure out that we'd typed the file's name wrong, or put the file in the wrong place.

We can do better. Take a look at Program 8.15. This version of the program checks to see if an error has occurred and prints out a more informative error message. This program does several new things, so let's look at them one by one.

Program 8.15: filecheck.cpp, with error messages

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
int main () {
  FILE *input;

  input = fopen( "nosuchfile.dat", "r" );
  if ( !input )  {
    fprintf ( stderr, "Error opening file: %s\n", strerror(errno) );
    exit(1);
  }

  // Do stuff, then close the file...
  fclose ( input );
}
```

[5] This error is generated when `fclose` tries to close `input`, which was never really set because the file couldn't be opened.

First of all, there are many kinds of errors that a function like fopen might encounter. For example:

- Maybe the file you're asking for doesn't exist.
- Maybe you don't have permission to read or create the file.
- If you're trying to create a new file, there might be no more room left on the disk.

In order to tell us what happened, the function identifies each of these conditions with an "error number". Notice that we've added "errno.h" to the list of #include statements at the top of the program. Among other things, this defines a new variable named errno that will always contain a number identifying the most recent error.

Having an error number is a step in the right direction, but words would be even better. That's what the strerror function does. It tells us, in plain English, what a particular error number means. strerror returns a character string that our program can print out to describe the error. In order to use strerror we need to add "#include <string.h>".

Finally, we need to have some way to stop the program when we see an error. There's often no point in continuing after something goes wrong, and doing so could even be dangerous. To stop a program immediately, we can use the exit function. It takes a single argument (an integer) that's passed along to the operating system to indicate whether the program finished successfully or died because of an error. A value of zero indicates success, and anything else means failure[6]. exit requires stdlib.h.

If we ran our improved program (with nosuchfile.dat still missing), it would say:

```
Error opening file: No such file or directory
```

That's much more informative than "Segmentation fault"! When writing programs, think about what might go wrong and try to deal with these situations gracefully.

[6] We won't make use of these exit values in this book, but they can be handy when writing "scripts" that run programs for you.

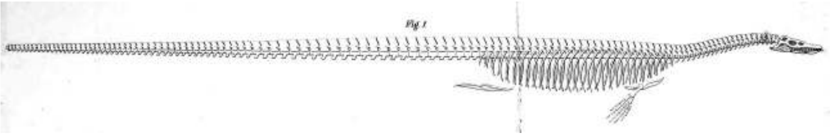


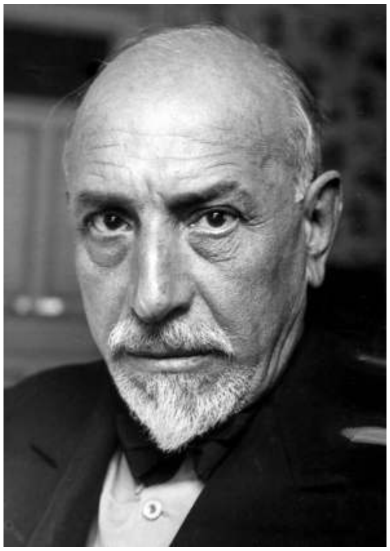
Figure 8.10: We all make mistakes. In 1890, palaeontologist Othniel Marsh humiliated his rival Edward Cope by pointing out that Cope had reconstructed the skeleton of *Elasmosaurus* with the head on the wrong end!

Source: Wikimedia Commons

8.15. Converting Characters to Numbers

As we noted in Section 8.4, the computer stores everything as ones and zeros, and it uses ASCII codes to store characters. For example, the ASCII code for an upper-case 'A' is 01000001. If we interpreted this as a binary number, it would be equal to the decimal number 65.

There's an ASCII code for each character on your keyboard, including all the numbers. The ASCII code for the digit '1' is 00110001. Interpreted as a binary number, this would be equivalent to the decimal number 49. Take a look back at Figure 8.3 to see the ASCII codes for some other digits.



Luigi Pirandello was the author of the 1921 play *Six Characters in Search of an Author*. I remember the 1976 PBS production, starring John Houseman and Andy Griffith(!).

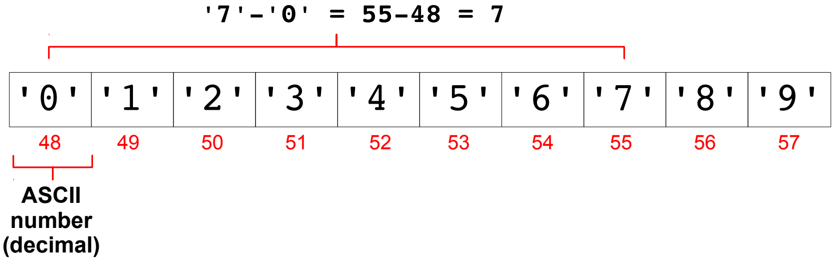
Source: Wikimedia Commons

On the other hand, computers store integer *numbers* as a binary representation of the number. For example, the number 1 would be stored as 00000001. Maybe you can see how this could create some confusion. As far as the computer is concerned, *character* '1' is completely different from the *number* 1.

Sometimes we'll need to convert a character that represents a digit into an actual number. How can we do that? The first clue is to notice that the ASCII codes for all of the digits in Figure 8.3 are sequential. If we converted these binary numbers into decimal, we'd see that '0', '1', '2', and '3' are represented by the numbers 48, 49, 50, and 51.

The second clue is provided by a feature of C that we haven't mentioned before: C is perfectly happy to do math with char variables. It just treats the character variable as though it had a value equivalent to the decimal representation of its ASCII code. So, the computer would see '1'+'2' as 49+50, giving a value of 99.

Using these two clues we can do a little math and determine the numerical value of a character. Take a look at the figure below.



If we want to find the numerical value of the character `'7'` we just need to subtract the character `'0'` from it. In a program, that might look like this:

```
int n;
char c = '7';
n = c - '0';
printf( "The numerical value of %c is %d\n", c, n );
```

But what about. . . ?

What if we have a multi-digit number represented as a string? For example, the string `"186282"`? In principle, we could go through it one digit at a time, converting each character into a number and multiplying it by the appropriate power of ten, then adding up all the results. This would be tedious though, and it seems like something we might need to do pretty often.

Fortunately, as we'll see in Chapter 9, C provides us with two functions that will do the work for us. They're named `atoi` and `atof`. The `atoi` function converts a string of digits into an integer. The `atof` function converts a string that might contain decimal points into a `double`. For example:

```
char ci = "12345";
char cd = "67.890";
int i;
double d;

i = atoi( ci );
d = atof( cd );
```

As we'll see in Chapter 9, these two functions come in very handy in one particular situation: Interpreting command-line arguments.

Let's look at an example that uses this trick.

8.16. Multiplicative Persistence

In Number Theory there's a fun property of numbers called *multiplicative persistence*[7]. Take the number 39, for example. It's represented by the two digits 3 and 9. If we multiply 3×9 we get another number, 27. Multiplying 2×7 gives 14. Multiplying 1×4 gives 4. Now we're down to just one digit after three steps: $39 \to 27 \to 14 \to 4$. We say

[7] See this YouTube video by Matt Parker on the Numberphile channel: https://www.youtube.com/watch?v=Wim9WJeDTHQ

that 39 has a multiplicative persistence of 3, meaning that we can do this procedure of multiplying the digits three times before we get to a single-digit number.

Try this with some other numbers. You'll find that most numbers have only a small persistence. 39 is actually the first one that gets as high as 3. The persistence of 77 is 4. The first number with a persistence of 5 is 679, and you have to go all the way to 6,788 to find a number that has a persistence of 6. Mathematicians think that no base-10 number has a multiplicative persistence greater than 11, but this remains unproven (although it's been checked for numbers up to $10^{20,000}$!).

Still I Persist in Wondering is the name of an excellent story collection by Edgar Pangborn.

Source: Goodreads

Let's write a program that tests the multiplicative persistence of a given number. Take a look at Program 8.16.

Program 8.16: mpersist.cpp

```
#include <stdio.h>
#include <string.h>
int main () {
  const int maxdigits = 10;
  char number[maxdigits];
  int length;
  int product;
  int i;

  printf ("Please enter a number, up to %d digits long: ", maxdigits-1 );
  fgets ( number, maxdigits, stdin );
  number[ strcspn( number, "\r\n" ) ] = '\0';

  length = strlen( number );
  while ( length > 1 ) {
          product = 1;
          for ( i=0; i<length; i++ ) {
            product *= number[i] - '0';
          }
          snprintf ( number, maxdigits, "%d", product );
          length = strlen( number );
          printf ( "%d %s\n", length, number );
  }
}
```

The program stores a number in a character array. This lets us easily get each digit of the number, since each digit is one element of the array. The program uses `strlen` to find the string's length. Notice that the "`while`" loop keeps going as long as `length` is greater than one. Each time around the loop, a "`for`" loop goes through all the digits of the number, converting each digit to its numerical equivalent by subtracting `'0'`. The variable named `product` keeps track of the product obtained by multipying the digits together.

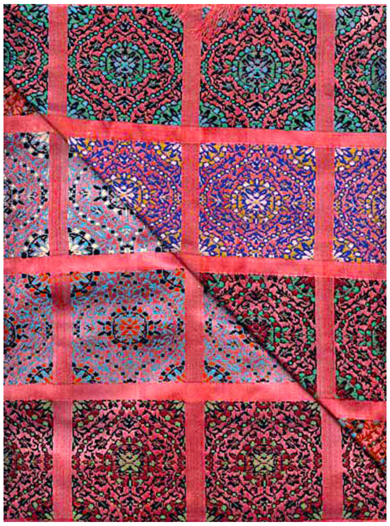
If we ran the program, we would see something like this:

```
Please enter a number, up to 9 digits long: 39
2 27
2 14
1 4
```

The first column is the number of digits, and the second column is the current product.

8.17. Pattern Matching

We've seen how `strcmp` lets us compare two strings to see if they're equal, but what if we want to know whether the string fits some fuzzier pattern? For example, we might want to know if the string begins with an upper-case letter, or we might want to check for any of the strings "y", "Y", "yes", or "YES".



Source: Wikimedia Commons

The GNU C compiler supports a powerful pattern-matching system called "Regular Expressions". Regular Expressions (sometimes called "regexp" for short) are used in many computer languages. A little knowledge about them will be useful no matter what language you use.

Regular Expressions are a way of specifying a pattern that you want to match. The pattern is written as a group of symbols that can represent particular characters, ranges of characters, or wildcards of various kinds that will match any character. The Regular Expression language is extensive, but here are some commonly-useful symbols and their meanings:

Symbol	**Meaning**
`.`	Match any single character.
`*`	Match zero or more of the preceding item.
`+`	Match one or more of the preceding item.
`?`	Match zero or one of the preceding item.
`{n,m}`	Match at least n, but not more than m, of the preceding item.
`^`	Match the beginning of the line.
`$`	Match the end of the line.
`[abc123]`	Match any of the enclosed list of characters.
`[^abc123]`	Match any character *not* in this list.
`[a-zA-Z0-9]`	Match any of the enclosed ranges of characters.
`this|that`	Match "this" or "that".
`\.`, `\*`, etc.	Match a literal "`.`", "`*`", etc.

Regexp patterns can get confusing very quickly, but here are some simple examples:

`^Y`	Match any string beginning with `Y`.
`^[Bb]ob`	Match any string beginning with `bob` or `Bob`.
`100$`	Match any string ending in `100`.
`^T.*day$`	Match `Tuesday`, `Thursday`, or any other string that begins with a `T` and ends with `day`.
`^data[0-9][0-9]\.dat`	Match `data01.dat`, `data02.dat`, or any other string with `data` followed by two digits and `.dat`.

Program 8.17 uses regular expressions to identify strings that begin with upper-case letters. A string like `Montana` would match, but `montana` wouldn't.

The program uses two functions to accomplish this: `regcomp` and `regexec`. The first function "compiles" a Regular Expression into an internal form that's easier for the computer to use. The second function uses this compiled Regular Expression to test a string. In this case, the Regular Expression we're using is `^[A-Z]`, which matches any string that begins with the upper-case characters `A` through `Z`.

Notice that we need to add `#include <regex.h>` in order to use these functions. `regex.h` also defines a new type of variable, `regex_t`, that's used for storing the compiled version of a Regular Expression.

A complete description of the `regcomp` and `regexec` functions is beyond the scope of this course, but Program 8.17 illustrates their basic usage. In this example, `regcomp` compiles our expression and stores the compiled version in the variable named `reg`. The "`REG_NOSUB | REG_EXTENDED`" argument we give `regcomp` just specifies a couple of options that you'll probably want to use.

The program gives the `regexec` function the compiled Regular Expression (stored in the variable `reg`) and the name of a string variable to test. The other three arguments we give `regexec` aren't really used in this case, but they should usually be set to the values shown here.

Program 8.17: match.cpp

```
#include <regex.h>
#include <stdio.h>
int main()
{
  regex_t reg;
  char string[100];

  printf ("Enter a word: ");
  scanf( "%99s", string );

  regcomp( &reg, "^[A-Z]", REG_NOSUB | REG_EXTENDED );
  if ( regexec( &reg, string, 0, NULL, 0 ) == REG_NOMATCH) {
    printf ("Doesn't match.\n");
  } else {
    printf ("\"%s\" Matches!\n", string);
  }
}
```

8.18. Conclusion

Character strings aren't particularly exciting, but it can be convenient to be able to use them in your programs. When doing so though, be careful not go past the end of your character arrays, and remember that you need to use `strcmp` to compare strings. If you stick to the best practices outlined above, you should be alright.

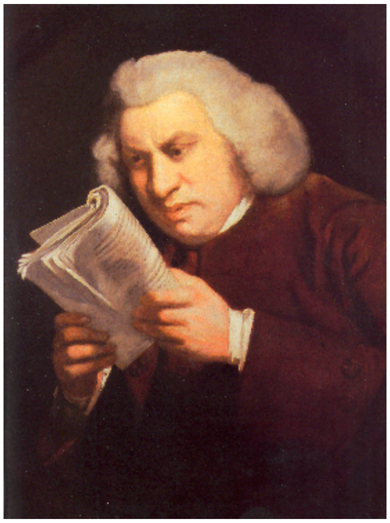
The author's Uncle Buster, playing a stringed instrument. Buster was a character. (Character. String. See what I did there?)

Practice Problems

1. Write a program called `dict.cpp` that asks the user for two words (reading them with `scanf`), and then tells you which word would come first in the dictionary. If the two words are the same, the program should tell you so. Assume the words are less than 100 characters long.

2. Write a program called `hiname.cpp` that asks users to enter their first and last names, on a single line like "Bryan Wright". Use a single `scanf` statement to read the user's names. Make the program then say "Hi", followed by the user's first name, like "Hi Bryan!".

3. Write a program called `getpoem.cpp` that asks users to enter the first line of their favorite poem. Let the line be up to 100 characters long. Make your program open a file named `firstlines.dat` and write the line into the file. Be sure to open the file for "appending", by giving `fopen` an `"a"` as its last argument[8]. Try running the program several times and entering different lines. If you look at `firstlines.dat` with *nano*, you should see all of the lines you've typed in.



In 1755 Samuel Johnson published his *A Dictionary of the English Language*. It remained the most respected English dictionary until more comprehensive dictionaries were published in the 20th Century.

Source: Wikimedia Commons

[8] See Chapter 5.

4. The following statement will get the current time (measured in seconds since January 1, 1970) and put it into an integer variable named `start`:

```
start = time(NULL);
```

Knowing this, write a program called `type1.cpp` that tests how fast a user can type the phrase "I love programming!". Make sure the program tells the user what to do.**Hints:**

- Use a character string at least 30 characters long to capture what the user types.
- Remember to add "`#include <time.h>`" for the `time` function
- Use `fgets` to read what the user types
- Check the time before typing and the time after typing, then look at the difference to find out how long it took. Tell the user how many seconds it took him or her to type the phrase.
- Don't bother to check whether the user typed the right thing. Assume the user is honest.

The stylish Olivetti *Valentine* typewriter, designed by Ettore Sottsass to be "sensual and exciting".

Source: Wikimedia Commons

5. After completing Problem 4 modify the program so that it uses a `for` loop to ask the user to type the phrase three times, then tells

the user his or her average speed. Give the speed in three ways: average number of seconds to type the phrase, number of characters per second, and number of words per minute (where a standard "word" is five characters or spaces). Be sure to allow for non-integer speeds like "1.5 characters per second". Call the new program `typing.cpp`.

Source: Wikimedia Commons

6. First, fetch a copy of Lewis Carroll's book *Alice in Wonderland* by typing either:

   ```
   wget http://tinyurl.com/y9nrg3xh
   ```

   or

   ```
   curl -L -O http://tinyurl.com/y9nrg3xh
   ```

   After you've downloaded the file, rename it by typing:

   ```
   mv y9nrg3xh alice.txt
   ```

   Then, write a program named `wordlength.cpp` that reads `alice.txt` and reports the average length of the words in the book. The program should define a large (say, 100-character-long) character string, then it should have a loop that repeatedly uses `fscanf` to read words from the file. Use the `strlen` function to find each word's length (see Section 8.6).

   Can you see why we often use 5 characters as the length of a standard "word" when measuring text?

   Hints: To find the average you'll need to first add up the lengths of all the words. I recommend you use a `double` variable to hold this sum. If your program tells you the average word length is exactly an integer, you've done something wrong.

7. After completing Problem 6 create a new version of your program that also makes a histogram that shows the distribution of word lengths. (See examples in Chapter 7.) To do this, you'll need to add an integer array that will keep track of how many words have a given length. Call this array `count` and make it 50 elements long. Each element of the array will contain the number of words that have a length equal to that bin's index. For example, `count[5]` will have the number of 5-letter words. At the end of the program, print out two columns showing the number of letters and how many words had that many letters. Call your new program `wordhist.cpp`.

Notice that, by making our array 50 elements long, we limit ourselves to words with a length between zero and 49 letters. Be sure your program checks the word length to make sure it isn't outside those limits.

If your program also prints out the average word length (as in Problem 6) make sure to put a # at the beginning of the line, so *gnuplot* won't be confused by it if you want to plot your results (see Figure 8.11).

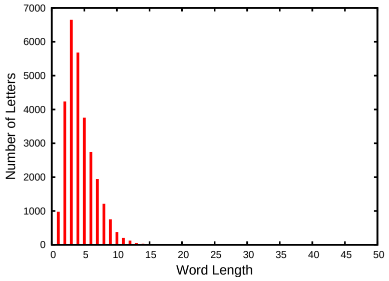


Figure 8.11: If you plot a histogram of the word lengths, you should see something like this.

8. Write a program named `charcount.cpp` that counts how many times each letter of the alphabet appears in a file full of text, treating upper- and lower-case letters as different. Start out by downloading a copy of *Alice in Wonderland* by Lewis Carroll. You can do this with one of the two commands below:

```
wget http://tinyurl.com/y9nrg3xh
```

or

```
curl -L -O http://tinyurl.com/y9nrg3xh
```

After you've downloaded the file, rename it for convenience by typing:

```
mv y9nrg3xh alice.txt
```

Your program should take advantage of the fact that, in C, a character is equivalent to the character's numerical ASCII code. For example, the character "A" is ASCII character number 65. If you have a character variable named `c`, you can get the numerical ASCII code for the character it contains by saying `(int)c` (that is, just "casting" the character as an `int`). These numbers are in the range from zero to 255.

Lewis Carroll, whose real name was Charles Lutwidge Dodgson, was also an accomplished mathematician who made significant contributions to that field.

Source: Wikimedia Commons

At the top of your program, create a 256-element array of integers named `count`, like this:

```
int count[256] = {0};
```

The `= {0}` is a trick we saw earlier in this chapter that sets all of the array elements to zero initially. Your program should contain a "`while`" loop that reads one character at a time from the file `alice.txt`. Each time a character is read, add 1 to the element of `count` that has an index corresponding to that character's ASCII code. You can do that with a line like this:

```
count[ (int)c ]++;
```

After the “while” loop is finished, the program should have two “for” loops to print its results. The first loop should print counts for character numbers 65 through 90, which corresponds to all the upper-case letters. The second loop should print counts for characters 97 through 122, the lower-case letters. Each line of the output should be printed like this:

```
printf ( "%c %d\n", i, count[i] );
```

where i is the character number. Notice that if we print an integer variable using %c the program will just print the character corresponding to that number. So, for example, if the file contained the character “A” 807 times, the program would print a line like this for that character:

```
A 807
```

After you’ve written your program and tested it, try redirecting its output into a file, like this:

```
./charcount > charcount.dat
```

Then you can use *gnuplot* to generate the graph in Figure 8.12. The *gnuplot* command to do this is:

```
plot "charcount.dat" using ($0):2:xtic(1) with impulses
```

There are two bits of magic here: First, ($0) tells *gnuplot* to use the *line number* as the x value. Second, xtic(1) tells *gnuplot* to use the values in column 1 of the data file as the labels on the x axis.

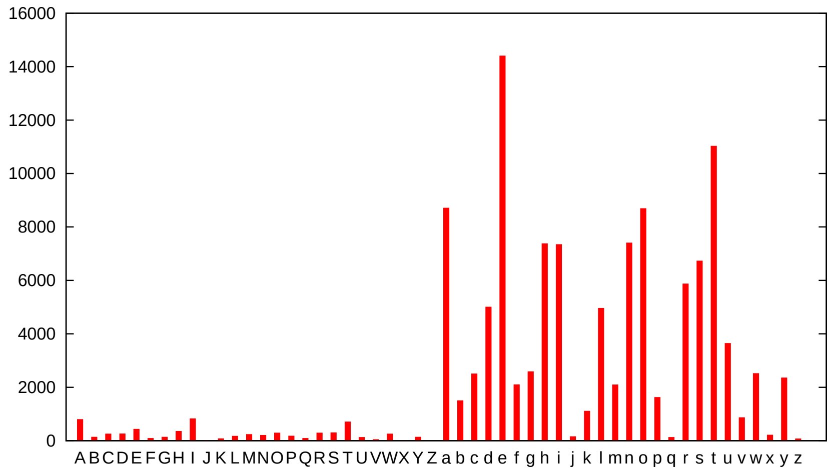


Figure 8.12: Number of each character seen in *Alice in Wonderland*. Notice that “e” is the most common, as is typical in English-language text.

9. First, fetch a copy of the file `unixdict.txt` by typing either:

```
wget http://wiki.puzzlers.org/pub/wordlists/unixdict.txt
```

or

```
curl -L -O http://wiki.puzzlers.org/pub/wordlists/unixdict.txt
```

(whichever works on your computer). This file contains a long list of over 25,000 English words. You can open the file with *nano* to see them.

Write a program named `longestword.cpp` that reads this file and finds the longest word. The program should print the word and its length. Use the `strlen` function to find each word's length (see Section 8.6)

Use the `strerror` function, as described above in Section 8.14, to print an error message if the file `unixdict.txt` can't be found.

Assume that no word is longer than 1,000 characters. Also (of course) assume that all the words have more than zero characters.

Hints: You'll need to define two character strings: one to hold the word you've just read from the file, and another to keep track of the word that has the maximum length so far. Also, you'll find it simpler if you do the following as soon as you read each word:

```
length = strlen(word);
```

then look at the value of `length` when deciding whether this word is longer than the current record-holder. Use an integer variable to keep track of the length of the current record-holder. the top of your program might look something like this:

```
char word[1000];
char maxword[1000];
int length;
int maxlength;
```

A dictionary from 1st-Century BCE Uruk, in Mesopotamia.

Source: Wikimedia Commons

10. Write a program named `linenumbers.cpp` that prints the contents of a file and writes a line number at the beginning of each line, starting with 1. The program should begin by asking the user for the name of the file they want to print, then open that file for reading. Next, the program should use a "`while`" loop to read each line from the file. You'll want to use `fgets`, because the lines will contain spaces. Each time you read a line, print the line's number, followed by a colon and a space, and then the content of the line. For example,

if you asked the program to print your hello.cpp program, it should print something like this:

```
Enter file name: hello.cpp
1: #include <stdio.h>
2: int main () {
3:      printf( "Hello world!\n" );
4: }
```

Use the following variables for reading the file name and for reading the contents of a line:

```
  char filename[128];
  const int maxline = 1024;
  char line[maxline];
```

Hint: When the fgets function reaches the end of the file, it returns a value that "while" interprets as false. Otherwise, the value returned by fgets looks true to "while".

11. A character string is just an array of characters. If a string named motto contains the text "I like science!", then we know that motto[0] has the value 'I', motto[7] has the value 's', and so forth. We can easily get to the individual characters that make up the string.

    Let's use this ability to help us write a program that does a simple math game. Start with a number like 137. Treat the digits of this number like three separate numbers, and add them together: 1+3+7 = 11. Now repeat this for the new number: 1+1 = 2. There's only one digit now, so there's nothing else we can do. The number we've ended up with is called the *digital root* of the original number.

    Write a program named digitalroot.cpp that asks the user for a number. Then the program should add the digits of that number together to get a new number, and keep repeating this process until it gets to a single-digit number. Print each of sums as you calculate it.

    Your program is going to have to do some conversion between characters (and character strings) and numbers. I recommend the following:

    - When you ask the user for the initial number, read this number into an `int` variable.

- When you get ready to add a number's digits, use the `snprintf` function to write the number into a character string. For example, you might have a character array named `c` that's defined like this:

  ```
  const int cmax=100;
  char c[cmax];
  ```

  Then you could write a number, `n`, into that string like this:

  ```
  snprintf( c, cmax, "%d", n );
  ```

- In order to add the digits stored the the characters of `c` you'll need to know how many digits there are. Remember that you can use the `strlen` function to tell you this.
- Finally, you'll need to convert each character (like '3') into an integer (like 3) in order to add up the digits. There's a simple trick to do this. The integer value of the character in `c[i]` is just `c[i] - '0'`. Can you figure out why this is true?

Try your program with a few numbers. Can you find one that takes more than two steps to get to a single digit?

9. *Functions*

9.1. Introduction

Despite what you may think after reading the preceding chapters, C is really a very minimal language with only a small vocabulary of about 32 words. This is one reason C has been so successful.

Functions allow you to extend the capabilities of the C compiler.

Different types of computer understand different binary instructions, so programs that run on each kind of computer need to be created by a compiler that knows that computer's instruction set. Because making a C compiler is relatively easy (compared to many other computing languages), C is often the first language available when a new type of computer is developed.

Even though the C language is simple, it's powerful because we can extend its abilities by adding "functions" to it. We've already used many of these: `printf`, for example, isn't part of the C language. It's a separate function that has been added. The same is true of the other reading and writing functions we've been using, and the math functions like `sqrt`. All of these are found in standard "libraries" of functions that are usually installed along with the C compiler. The functions in these libraries are themselves written in C. They're essentially pre-compiled snippets of programs, ready to be plugged in where you need them.

*	auto	*	int
*	break		long
	case		register
*	char	*	return
*	const		short
*	continue		signed
	default		sizeof
*	do	*	static
*	double		struct
*	else		switch
	enum		typedef
	extern		union
	float		unsigned
*	for	*	void
	goto		volatile
*	if	*	while

Figure 9.1: The 32 words of the C language, with an asterisk beside those we've already covered or will cover in this chapter.

Just as you can extend a house by building an extra room, you can build functions that extend the C compiler's capabilities. In this chapter we'll learn how functions work, and see how to create functions of our own.

C's functions let us define simple words to do complicated things. This is especially useful when we have to do a complicated operation over and over again, but it can help us in other ways too. Functions can be re-used in other programs, and using functions can help you avoid programming mistakes.

9.2. What's a Function?

Let's start out by reviewing the kind of functions you've used in math class. Figure 9.2 shows the mathematical function $f(x) = x^2 + 3$. The function is like a machine that takes some raw materials and processes them to produce an output. The function's raw materials are its *arguments*. The function in Figure 9.2 takes one argment, which we've called x here. We could just as easily have written $f(y) = y^2 + 3$. The name we give the argument doesn't matter. It's just a placeholder.

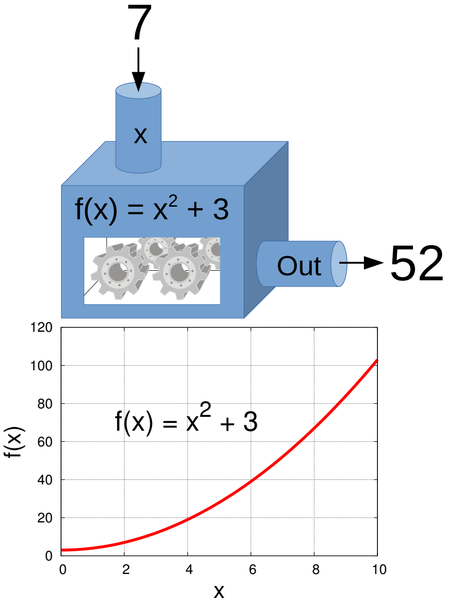


Figure 9.2: You're probably familiar with mathematical functions. A function takes some arguments (inputs), performs some operations on them, then spits out a result.

When we write $f(x) = x^2 + 3$ we're *defining* a function. We've given our function a name, f, we've specifed that it takes one argument (x), and we've said what the function does with that argument to produce an output (square the argument and add three to it). If we put in the value 7, as in Figure 9.2, we'll get out the value 52. We could try a range of different input values and plot the corresponding output values on a graph, as in the lower part of Figure 9.2.

Functions can have more than one argument. Consider the function $g(x, y)$ shown in Figure 9.3. This function takes two arguments (x and y) and produces an output that combines them in a particular way. Functions can have any number of arguments.

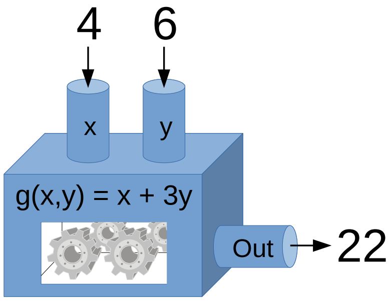


Figure 9.3: A function that takes two arguments, x and y. Given x=4 and y=6 as arguments, the function's output would be 22.

Functions can also make use of *other* functions, as illustrated in Figure 9.4. Here, the function $h(x)$ is defined to be $h(x) = i(x) + 5$, where $i(x)$ is another function, defined as $i(x) = 3x^2$. If we gave $h(x)$ an input of x=2, it would find $i(2)$ = 3×4 = 12, and then add five to this to find that $h(2)$ = 17.

A function in a C program has all the properties we described above:

- A function has a *name*
- The function takes *arguments* and uses them to produce an *output*
- The behavior of a function is described by *defining* the function
- Functions can have *any number* of arguments (in fact we'll see that C function sometimes take no arguments at all!)
- Functions can use *other functions*

As we'll see below, C functions also have some properties that aren't present in mathematical functions.

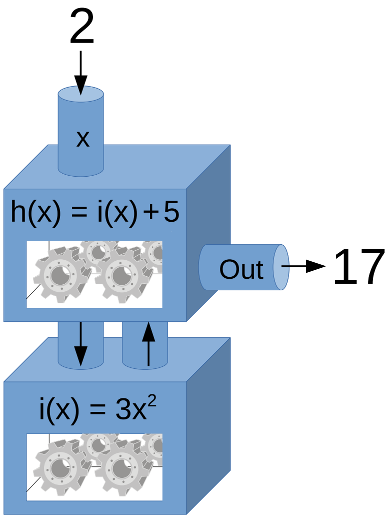
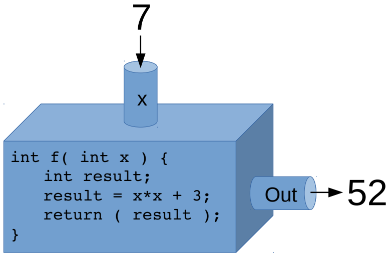


Figure 9.4: The function $h(x)$ shown above uses another function $i(x)$.

Let's look at how we might define a function in a C program. Figure 9.5 shows a C function that takes an input value (an integer we call x) and produces an output value that's equal to x*x + 3. This is analogous to the mathematical function we saw in Figure 9.2.



```
int f( int x ) {
    int result;
    result = x*x + 3;
    return ( result );
}
```

Figure 9.5: A C function named f that does the same thing as the mathematical function shown in Figure 9.2.

Program 9.1 shows how we might insert this function at the top of a program, and use it to print some values of the function for various values of its argument. We could plot the program's output with *gnuplot* to create a graph like the one shown in Figure 9.2.

Program 9.1: funcfun.cpp

```
#include <stdio.h>
int f ( int x ) {
  int n;
  n = x*x + 3;
  return ( n );
}

int main () {
  int i;
  for ( i=0; i<10; i++ ) {
    printf ( "%d %d\n", i,  f(i)  );
  }
}
```

Function definition

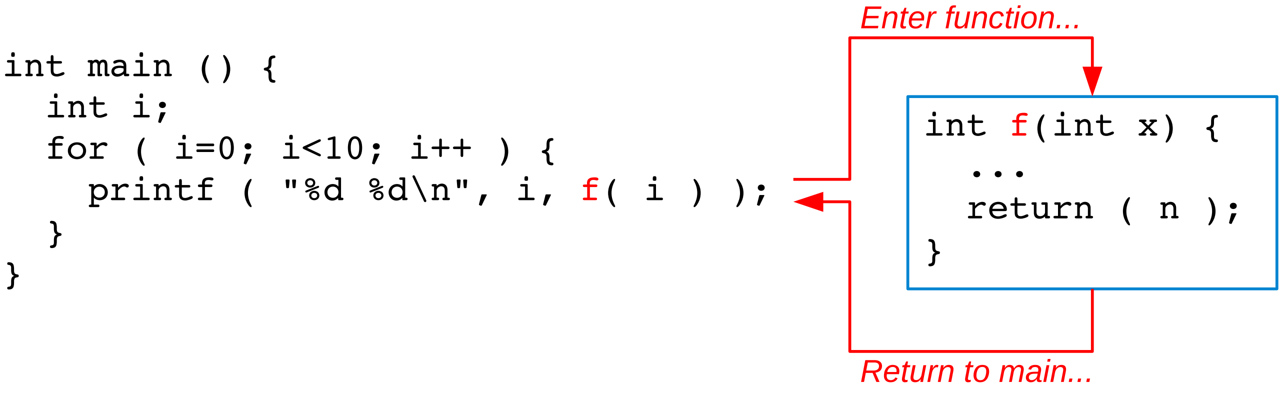
Using the function

This looks different from anything we've written before. We've added a new section above int main(). The new section defines a function named f. It says that the function accepts one int argument, and returns an int value. We then use this new function inside main(), in our printf statement.

You'll probably notice that the first line of our function definition looks an awful lot like the int main() statement that we've been using in all of our programs. That's no coincidence. main is a function just like sqrt, printf, or our new f function. It turns out that, in C, almost everything is inside of some function. When we run a C program, the computer looks for a function named "main" and does whatever that function tells it to do. We'll see later that we can even give arguments to main, as we do with other functions.

Also notice that we've defined our new function *above* main. The compiler needs to know about a function before we can use it. One way to ensure this is to define new functions at the top of the program. We'll see another way to do this later, in Chapter 11, where we'll find out that the line `#include <stdio.h>` tells the compiler about functions like `printf` and `scanf`.

When we use a function in a program, it's as though the program takes a detour into the function and then comes back again with a value:



```
int main () {
  int i;
  for ( i=0; i<10; i++ ) {
    printf ( "%d %d\n", i, f( i ) );
  }
}
```

```
int f(int x) {
  ...
  return ( n );
}
```

Figure 9.6: The "flow" of the program travels into the function, and then comes back with a result.

Exercise 46: First Function

- Create, compile, and run Program 9.1. Redirect the program's output into a file by running it like this:

  ```
  ./funcfun > funcfun.dat
  ```

- Then use *gnuplot* to plot the program's output, using the *gnuplot* command:

  ```
  plot "funcfun.dat" with lines
  ```

  Does your result look like Figure 9.2?

- Now modify your program so that $f(x) = x^2 + 20$. Compile the program, and run it like this to produce a second data file (`funcfun2.dat`):

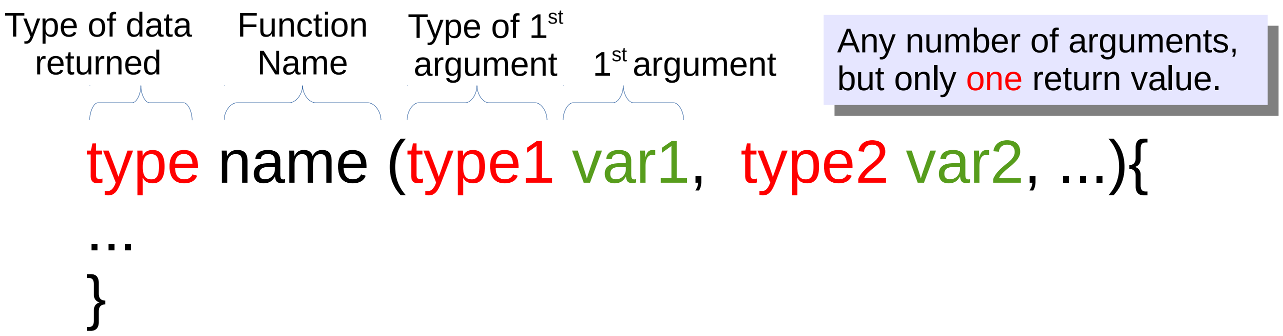
  ```
  ./funcfun > funcfun2.dat
  ```

  then use the following *gnuplot* command to plot both files on the same graph:

  ```
  plot "funcfun.dat" with lines, "funcfun2.dat" with lines
  ```

9.3. Function Anatomy

The anatomy of a function definition looks like Figure 9.7. First we need to specify what type of value the function will return. This can be any of the types we use for variables: `double`, `int`, or `char`, for example.

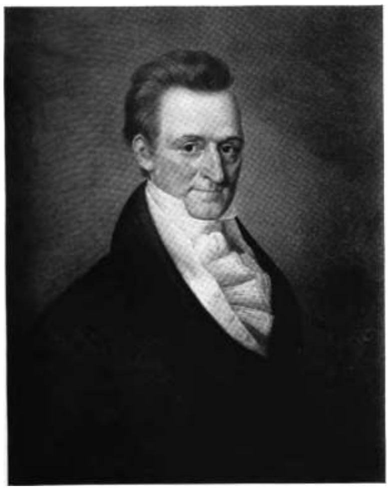


```
type name (type1 var1,  type2 var2, ...){
...
}
```

Figure 9.7: The general form of a function definition.

The return value of a C function is like the value you get when you evaluate a function in algebra. The C expression `sqrt(4.0)`, for example, would return the value 2.0. The type of value returned by `sqrt` is a `double`.

By defining the type of value the function will return, you make it possible for the C compiler to check whether you're putting that value into an appropriate variable. If I write a statement like "`x = sqrt(4.0);`" the compiler will check to see if `x` is a `double` variable and give me a warning or an error message if it isn't.



Return J. Meigs, Jr., Governor of Ohio, US Postmaster General, and US Senator. As far as I know, C's '`return`" statement wasn't named for him, nor he for it.

Source: Wikimedia Commons

Next we give the new function's name. This must be different from the name of any other function in your program. Function names can contain letters (upper- or lower-case), numbers and underscores. As with variables (see Chapter 2), it's best to start the name of your function with a letter.

After the function's name, we list any arguments and their types. Our $f(x)$ function takes just one argument, and it's an integer. When we use a function in our program, the C compiler checks to make sure we're giving it the right number of arguments, and that the arguments are of the right type. If we've done something wrong, the compiler gives us a warning or an error message.

At the end of our function, as in our $f(x)$ function, we can optionally return a value, but we aren't obligated to return anything. Sometimes a function just does something without returning a value. For example, we might want a function that just prints some text. If a function doesn't return a value, we specify the function's type as "`void`", like this:

```
void howLong(int hours, int mins, int secs){
  printf("This class is %d seconds long\n",
          hours*3600 + mins*60 + secs);
}
```

Functions that *do* return a value use the `return` statement to do so. In our $f(x)$ example, the statement "`return(n)`" says that the function is done, and sends its result, `n`, back to the `main` function. Functions can only return one value.

Functions don't need to have any arguments, either. The `rand` function is an example of this. When defining a function that takes no arguments, just put an empty pair of parentheses after the function name.

Finally, functions can't be defined inside other functions. We couldn't, for example, define a new function *inside* `main`.

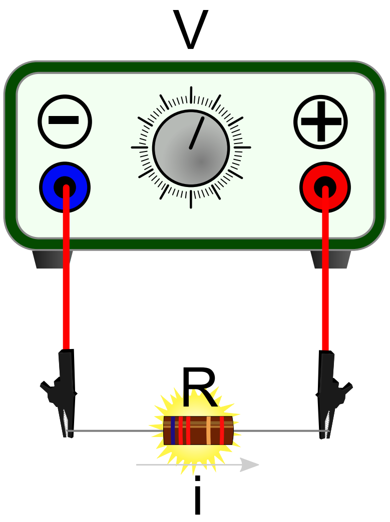


Figure 9.8: An adjustable voltage source is connected to a resistor, causing current to flow through the resistor.

9.4. Functions that Use Other Functions

Consider the apparatus shown in Figure 9.8. Ohm's law tells us that the current (which we represent by the symbol i) flowing through the resistor is given by:

$$i = V/R$$

where V is the voltage across the resistor and R is the resistance. Another law (Joule's Law) tells us that the power output of the resistor (which we represent by p) is given by:

$$p = i^2 R$$

The power is a measure of how fast the resistor is emitting energy, mostly in the form of heat. When we run a current through a resistor, the resistor heats up.

If we know the voltage and resistance, we can calculate the current, and then we can use the current to calculate the power. If we measure resistance in *ohms*, voltage[1] in *volts*, and current in *amperes*, the power we calculate will be given in units of *watts*.

Let's write a program that calculates the power output of the resistor at various voltage settings. The result might look like Program 9.2.

Georg Simon Ohm (left), Alessandro Volta (center), and André-Marie Ampère, for whom the units of resistance, electrical potential, and current are named.

Source: Wikimedia Commons, 1, 2, 3

[1] also called *electrical potential*

Program 9.2: ohm.cpp

```
#include <stdio.h>

double current ( double v, double r ) {
  return ( v/r );
}

double power ( double v, double r ) {
  double i, p;
  i = current ( v, r );
  p = i*i*r;
  return ( p);
}

int main () {
  double r = 100; // ohms.
  double vmin = 0; //volts.
  double vmax = 12; //volts.
  double v, p, vstep;
  int n;

  v = vmin;
  vstep = (vmax - vmin)/100.0;
  for ( n=0; n<100; n++ ) {
    p = power ( v, r );
    printf ( "%lf %lf\n", v, p );
    v += vstep;
  }
}
```

Notice that we've defined two functions, `current` and `power`. The `current` function tells us how much current will flow through the resistor when a given voltage is applied across it. It takes two arguments, `v` and `r`, and returns a value for the current. Because this is a very simple function (it just divides `v` by `r`) we can do the calculation right in the `return` statement. The `current` function just has one line in it!

But what about the `power` function. Shouldn't it have current as one of its arguments, instead of voltage? Sure, we could do it that way, but we want our program to tell us the power for a given voltage, so why not write our `power` function so that it does the calculation for us? Here we've written the `power` function so that it takes voltage and resistance as arguments, then internally uses the `current` function to calculate the current, before going on to calculate the power and return that.

This makes our `main` program very simple. We just loop through several voltage values and use the `power` function to find the power value at each voltage. The program assumes the resistance is 100 ohms. The program starts at the voltage `vmin` and goes up to the voltage `vmax` in 100 steps. Notice that we calculate the size of each voltage step (`vstep`) before starting the loop, and then add `vstep` to the voltage each time we go around. If we used *gnuplot* to plot the program's output, we'd see a graph like Figure 9.9.

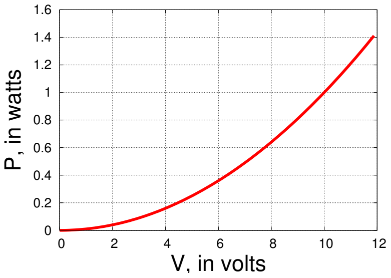


Figure 9.9: Power versus voltage for a 100 ohm resistor.

When you buy a resistor, you need to pay attention to the resistor's *power rating*. Some resistors can only tolerate a power output of $\frac{1}{8}$ watt. Trying to increase the power beyond that would cause the resistor to burn or melt. Resistors that can tolerate more than one watt are often called *power resistors*. Based on our program's output (as graphed in Figure 9.9) we'd need a power resistor that can tolerate at least 1.4 watts if we intend to put 12 volts across it.

Three high-power 100 ohm resistors, with power ratings of 10, 50, and 100 watts. Each has an aluminum case with cooling fins to help dissipate heat.

Source: Wikimedia Commons

Exercise 47: Your Volt Counts!

Create, compile and run Program 9.2. Send the program's output into a file and plot the data using *gnuplot*.

9.5. Variable Scope

If we run two different programs, we don't expect that the variables in one program will interfere with the variables in the other. It would be perfectly OK if one program had an `int` variable named `number` and the other program had a `double` variable with the same name. Variables don't affect things outside the program they're in. A programmer might say that the "scope" of a variable doesn't extend outside the program.

A scope of a different kind: UVa's own Professor Kathryn Thornton replaces solar panels on the Hubble Space Telescope.

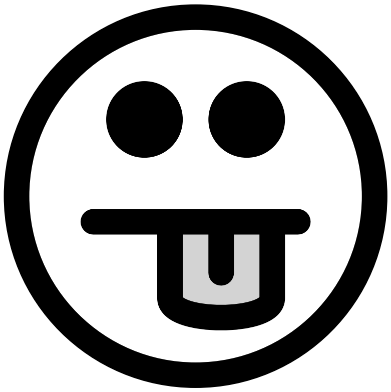
Source: Wikimedia Commons

In fact, in C, the scope of a variable might not even extend to other functions in the *same program*. Each variable in a C program has either a "local" or a "global" scope. All of the variables we've seen so far have local scope. This means that they can only be used inside the function where they're defined. Outside of that function, it's as though these variables don't even exist. (See Figure 9.10 on Page 320.)

The scope of a variable is determined by where it's defined. Variables defined inside a function are local to that function. Take a look at Program 9.3.

Program 9.3: scope.cpp (This won't work)

```
#include <stdio.h>

void printstuff () {
  printf ( "The value of n is %d\n", n );
}

int main () {
  int n = 100;
  printstuff();
}
```



If you tried to compile this program, *g++* would say:

```
scope.cpp: In function 'void printstuff():
scope.cpp:4: error: 'n' was not declared in this scope
```

The variable `n` is only defined inside `main`. As far as the `printstuff` function knows, this variable doesn't even exist.

On the other hand, variables defined outside of any function are *global*. (See Figure 9.10.) They can be used anywhere in your program. Here's a modified version of the program above. All we've done is move one line:

Program 9.4: scope.cpp, with a global variable

```
#include <stdio.h>

int n = 100;

void printstuff () {
  printf ( "The value of n is %d\n", n );
}

int main () {
  printstuff();
}
```

The variable `n` now has a global scope, meaning that every function in your program has access to it. The program will now compile, and do what you expect.

But what happens if you define a global variable, and then define a local variable with the same name? In that case, the local variable takes precedence. Figure 9.10 illustrates this. The function `func2` defines a `double` variable named `height`, even though there's already a global `int` variable with the same name. Inside `func2`, the name `height` will always refer to the local `double` variable, and the global variable of the same name will be inaccessible.

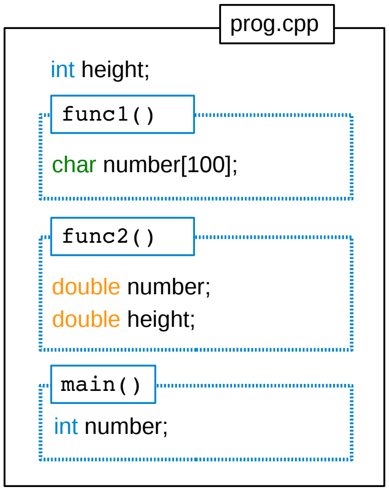


Figure 9.10: The three variables named `number` in `prog.cpp` are completely independent. Each one only exists inside the function where it's defined. Also notice that the `int` variable named `height` has a *global* scope, and can be used by any function. The function `func2`, however, *overrides* the global `height`, replacing it with a local `double` variable that only exists within that function.

It might be tempting to make all of your program's variables global, but avoid this temptation. In general you should use global variables sparingly. If many functions can change a variable's value it's very difficult to keep track of what's going on. It's much better to pass values to functions explicitly, via arguments, rather than to define them globally. For clearer code, it's best to restrict variables to the smallest possible scope. That being said, let's look at how global variables might profitably be used in a program.

9.6. Using Global Variables

Imagine that a rock is dropped from a balloon floating 1,000 meters above the ground, and falls under the influence of earth's gravity. We'll assume that the balloon is close enough to the earth so that we can use a constant value of $g = 9.8m/s^2$ for the rock's acceleration[2]. After some time, t, the rock's speed will be:

$$v(t) = gt$$

and it will be at a height h, where:

$$h(t) = 1000 - \frac{1}{2}gt^2$$

Program 9.5 tracks the falling rock for ten seconds. Every hundredth of a second it prints out the rock's current velocity and height. Two functions named `velocity` and `height` calculate those quantities. Notice that both functions are so trivial that they only contain a `return` statement. Both functions need to know the acceleration of gravity, so we store this in a global variable named `g`.

[2] We'll also ignore the effects of air resistance.

Program 9.5: falling.cpp

```
#include <stdio.h>

double g = 9.8;  // meters per second^2.

double velocity ( double t ) {
  return ( g*t );
}
double height ( double t ) {
  return ( 1000 - 0.5*g*t*t );
}

int main () {
  double t = 0; // elapsed time, in seconds
  int i;

  for ( i=0; i<1000; i++ ) {
    t += 0.01;
    printf ( "%lf %lf %lf\n", t, velocity(t), height(t) );
  }

}
```

Exercise 48: I've Fallen and I Can't Get Up!

Create and compile Program 9.5, then run the program like this:

```
./falling > falling.dat
```

The resulting file should contain three columns representing time, velocity and height. Now plot the height data by starting *gnuplot* and telling it:

```
plot "falling.dat" using 1:3 with lines
```

The phrase `using 1:3` tells *gnuplot* to use the first column (time) for the horizontal values on the graph, and the third column (height) for the vertical values. Does your plot look like Figure 9.11? What does a graph of velocity (instead of height) versus time look like?

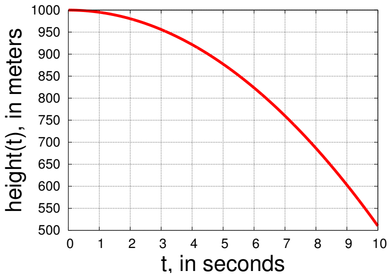


Figure 9.11: The height of a falling stone dropped from 1,000 meters, as a function of time.

9.7. Multiple Returns

Imagine that you're a fighter pilot who's been asked to fly his plane along a very specific (and quite odd) path. After you take off, you're supposed to rise steadily to a height of 1,000 meters, and then fly sinusoidally up and down for a while to evade enemy fire. After that, you're supposed to level off and fly at a constant height.

From the ground, your flight might look like Figure 9.12.

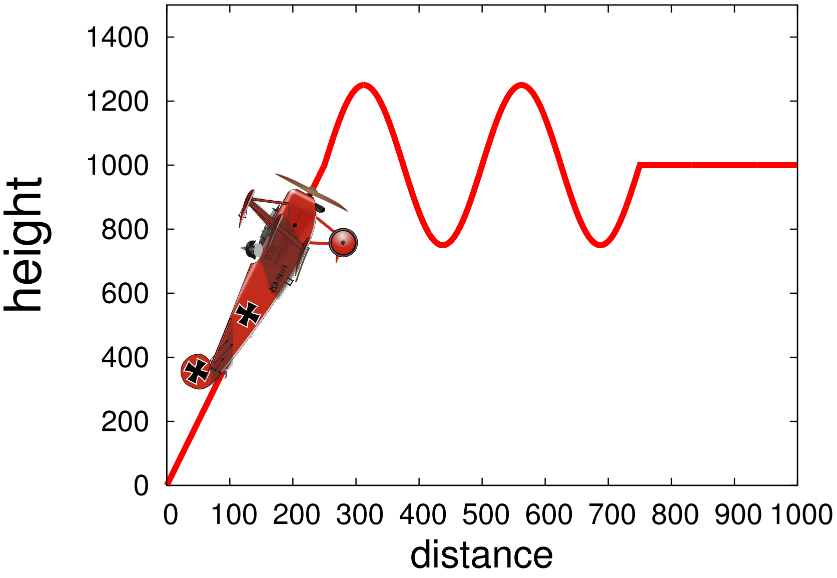


Figure 9.12: Our quirky flight path, which rises linearly for the first 250 meters, undulates for the next 250 meters, and then levels off. Fasten your seatbelts!

Can we write a function that tells us the plane's height as a function of how far the plane has travelled horizontally? One complication is the fact that our flight path has three distinct parts: takeoff, evasion, and cruising. As we see from Figure 9.12 each of the first two parts covers a horizontal distance of 250 meters.

We might consider writing a function that has an "if" statement, like this:

```
if ( x < 250 ) {
        // Takeoff
        ...
} else if ( x >= 250 && x < 750 ) {
        // Evasion
        ...
} else {
        // Cruising
        ...
}
```

The real "Red Baron", Manfred von Richthofen, the German WWI flying ace who flew a red Fokker triplane.

Source: Wikimedia Commons

Where x is the horizontal distance the plane has travelled.

Using such an "if" statement, we could define a "height" function like this:

Program 9.6: redbaron.cpp

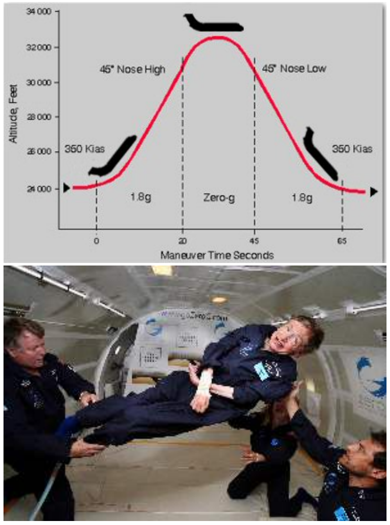
```
#include <stdio.h>
#include <math.h>

double height ( double x ) {

  if ( x < 250 ) {
    return( 1000 * x / 250 );
  } else if ( x >= 250 && x < 750 ) {
    return( 1000 + 250 * sin ( 2 * M_PI * (x-250) / 250 ) );
  } else {
    return( 1000 );
  }

}

int main () {
  int i;
  double x = 0;

  for ( i=0; i<1000; i++ ) {
    x += 1.0;
    printf ( "%lf %lf\n", x, height( x ) );
  }
}
```

Notice that the height function contains more than one return statement. That's OK. The function will use whichever return statement is appropriate, based on the value of x. It's perfectly alright to have a function use different return statements in different circumstances.

It's important to remember that a return statement ends the work done by a function. For example, if we had two lines like these in a function:

```
return ( 1 );
return ( 2 );
```

The function would always return a value of 1, since the first return would tell the function to stop working and return a value.



Aircraft sometimes do fly along odd trajectories. NASA's "Vomit Comet" creates zero-gravity conditions by temporarily flying along a parabolic path. Commercial ventures like "Zero Gravity Corporation" now use the same technique offer the experience to non-astronauts, like Physicist Stephen Hawking.

Source: Wikimedia Commons, Wikimedia Commons

9.8. Circus Physics

Consider the situation depicted in Figure 9.13. A circus performer, "*La Femme Melinite*", is launched from a cannon and flies throught the air, landing in a pool some distance away. As her manager, we'd like to make sure she doesn't miss, so we need to tell the roustabouts where to put the pool for a given cannon angle and initial velocity.

Source: Wikimedia Commons

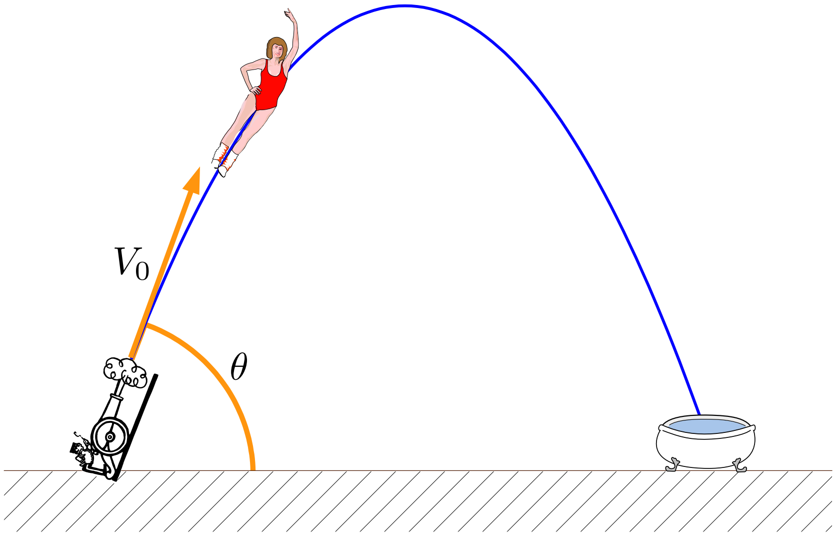


Figure 9.13: A human cannonball, launched at an intial velocity V_0, at an angle θ from the horizontal.

Fortunately, we've had some Physics classes so we know how to find the answer mathematically. The roustabouts, on the other hand, are all English majors. We need to write a computer program that they can use to find out where to put the pool each time they set up the circus.

The program will need to incorporate the mathematical facts of the problem. For example, we know that the total "time of flight" will be given by Equation 9.1, using the y-component of the initial velocity (see Figure 9.14).

$$t_{pool} = \frac{2V_{0y}}{g}, \; g = \text{the acceleration of gravity} \tag{9.1}$$

In our program, we could write a function that does the calculation in Equation 9.1. It might look like this:

```
double time_of_flight ( double v0, double angle ) {
  double t;
  t = 2.0 * v0 * sin(angle) / g;
  return ( t );
}
```

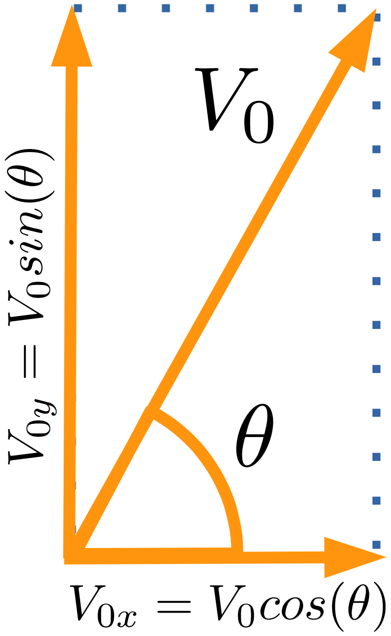
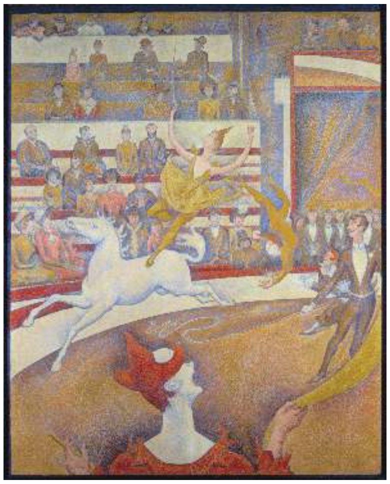


Figure 9.14: Using a little vector math, we can find the x and y components of V_0.

Program 9.7 is a little program that uses this function to tell us the time of flight. Notice that we've used a global variable, `g`, to hold the value of the acceleration of gravity. This value will be needed by several of the functions we'll be writing.



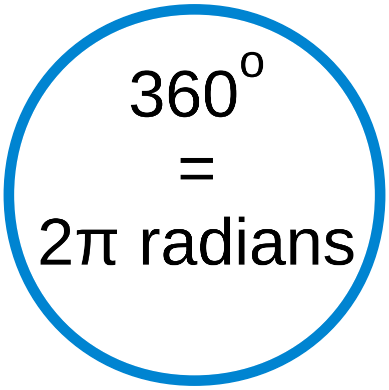
The Circus, by Georges Seurat (1891)

Source: *Wikimedia Commons*

Program 9.7: cannon.cpp

```
#include <stdio.h>
#include <math.h>

double g = 9.81; // Acceleration of gravity.

double time_of_flight ( double v0, double angle ) {
  double t;
  t = 2.0 * v0 * sin(angle) / g;
  return ( t );
}

int main () {
  double vinit;
  double theta;

  printf ( "Enter angle, in radians: " );
  scanf ( "%lf", &theta );
  printf ( "Enter velocity, in m/s: " );
  scanf ( "%lf", &vinit );

  printf ( "Time of flight is %lf sec.\n",
           time_of_flight( vinit, theta ) );
}
```

When we show this program to the roustabouts we're disappointed to find that they don't know how to measure angles in radians. No problem, though. We'll write a function that converts degrees into radians, and let them enter the angle in degrees:



```
double to_radians ( double degrees ) {
  return ( 2.0 * M_PI * degrees / 360.0 );
}
```

The `to_radians` function just contains one line (a return statement) and doesn't even define any variables. Now, after our program reads the angle, we can convert it into radians by saying "`theta = to_radians(theta)`".

Our Physics education also tells us how to find the maximum height of

La Femme Melinite's trajectory. (We want to make sure she doesn't hit the canvas of the Big Top!)

A circus tent.

Source: Wikimedia Commons

$$t_{peak} = \frac{t_{pool}}{2} \tag{9.2}$$

$$h = V_{0y} t_{peak} - \frac{1}{2} g t_{peak}^2 \tag{9.3}$$

This lets us write a function `max_height` to tell us how high our human cannonball will go.

```
double max_height ( double v0, double angle ) {
  double tpeak;
  double h;
  tpeak = time_of_flight( v0, angle ) / 2.0;
  h = v0*sin(angle)*tpeak - g*tpeak*tpeak/2.0;
  return ( h );
}
```

The last and most important thing we're interested in is the horizontal distance she will travel. That's given by Equation 9.4.

$$d = V_{0x} t_{pool} \tag{9.4}$$

We can express this in C as follows:

Source: Wikimedia Commons

```
double range ( double v0, double angle ) {
  double d;
  d = v0 * cos(angle) * time_of_flight( v0, angle );
  return ( d );
}
```

Putting all of these functions together with our earlier program, we get Program 9.8.

Program 9.8: cannon.cpp, with distance and height

```
#include <stdio.h>
#include <math.h>

double g = 9.81; // Acceleration of gravity.

double to_radians ( double degrees ) {
  return ( 2.0 * M_PI * degrees / 360.0 );
}

double time_of_flight ( double v0, double angle ) {
  double t;
  t = 2.0*v0*sin(angle)/g;
  return ( t );
}

double max_height ( double v0, double angle ) {
  double tpeak;
  double h;
  tpeak = time_of_flight( v0, angle ) / 2.0;
  h = v0*sin(angle)*tpeak - g*tpeak*tpeak/2.0;
  return ( h );
}

double range ( double v0, double angle ) {
  double d;
  d = v0 * cos(angle) * time_of_flight( v0, angle );
  return ( d );
}

int main () {
  double vinit;
  double theta;

  printf ( "Enter angle, in degrees: " );
  scanf ( "%lf", &theta );
  theta = to_radians( theta );

  printf ( "Enter velocity, in m/s: " );
  scanf ( "%lf", &vinit );

  printf ( "Time of flight is %lf sec.\n",
           time_of_flight( vinit, theta ) );

  printf ( "Max height is %lf meters.\n",
           max_height( vinit, theta ) );

  printf ( "Range is %lf meters.\n",
           range( vinit, theta ) );
}
```

Let's try our program out. Imagine that our flying lady is launched at a speed of 60 miles per hour (the world record for a human cannonball was set by someone travelling at about 70 mph). That's approximately equal to 27 meters per second. If the cannon is pointing upward at an angle of 45°, our program tells us the following:

```
Enter angle, in degrees: 45
Enter velocity, in m/s: 27
Time of flight is 3.892331 sec.
Max height is 18.577982 meters.
Range is 74.311927 meters.
```

The Circus, Charles Demuth (1917)

Source: Wikimedia Commons

Note that the Big Top will need to be at least 20 meters tall: as high as a six-story building! Also notice that she'll be in the air for almost four seconds. That's not bad, considering that riders in the Vomit Comet get only 25 seconds of weightlessness during each of the airplane's parabolic leaps (Figure 9.7).

9.9. Passing Values to Functions

The argument names we use when defining a function become local variables inside that function, just like any other local variables that we might define inside it. We can demonstrate that with Program 9.9.

Program 9.9: passing.cpp

```
#include <stdio.h>

void changenum ( int number ) {
  printf ( "Multiplying %d by 1000...\n", number );
  number = number * 1000;
  printf ( "...the result is %d\n", number );
}

int main () {
  int mynum = 1234;
  changenum( mynum );
  printf ( "My number is now %d\n", mynum );
}
```

If we ran this program, we'd see something like this:

```
Multiplying 1234 by 1000...
...the result is 1234000
My number is now 1234
```

What's going on? Why isn't the last number `1234000`?

When we use the function, the computer copies the values of the arguments we give it into the internal, local variables named in the function definition. In Program 9.9, the value of `mynum` (1234) gets copied into the `changenum` function's local variable `number`. Nothing we do to `number` has any affect on the variable `mynum` in `main`.

In fact, the local variables inside functions are, by default, non-existent whenever the function isn't being used. Let's take a look at the computer's memory before, during, and after using the `changenum` function. Figure 9.15 shows what we might see.

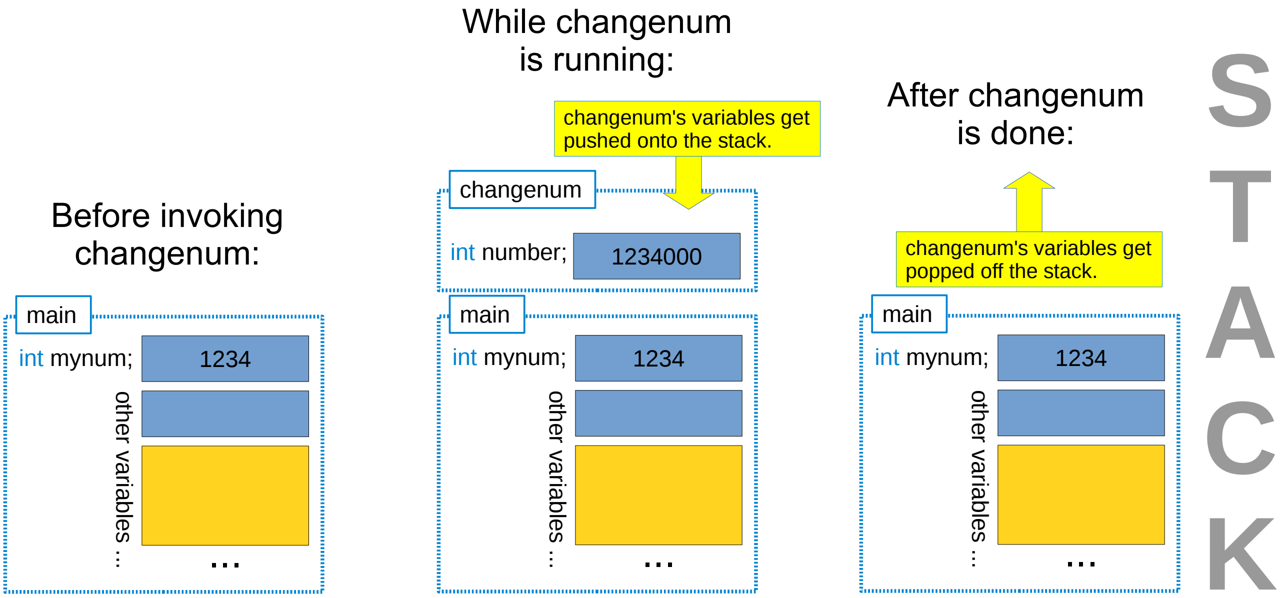


Figure 9.15: The stack before, during, and after using the `changenum` function.

When the function begins, the computer allocates some memory at the top of the stack for each of the function's local variables. When the function finishes, the allocated memory is freed up for other uses (perhaps by the next function that's used). A function's local variables literally disappear when they're not in use.

We can actually see that `mynum` and `number` are stored in different locations by asking our program to print the memory address of each of these variables. Program 9.10 does that by using `&number` and `&mynum` to get the memory addresses, and C's special placeholder for printing memory addresses, "`%p`".

Program 9.10: passing.cpp

```
#include <stdio.h>

void changenum ( int number ) {
  printf ( "number is at %p\n", &number );
  printf ( "Multiplying %d by 1000...\n", number );
  number = number * 1000;
  printf ( "...the result is %d\n", number );
}

int main () {
  int mynum = 1234;
  printf ( "mynum is at %p\n", &mynum );
  changenum( mynum );
  printf ( "My number is now %d\n", mynum );
}
```

If we run Program 9.10 we'll see something like this[3]:

```
mynum is at 0xbfd36ccc
number is at 0xbfd36cb0
Multiplying 1234 by 1000...
...the result is 1234000
My number is now 1234
```

9.10. Static Variables

As we saw in the preceding section, a function's local variables disappear when the function isn't in use, and are re-created each time we use the function. What if we want to save the value of one of these variables? Maybe, for example, we'd like to have a counter that tells us how many times the function has been called. We could accomplish that with a global variable, but there's also another way to do it.

Source: *Wikimedia Commons*

Take a look at Program 9.11. It uses the word "static" to tell the compiler that we want to retain the value of a variable even when the function isn't being used. Static variables don't live on the stack with other variables. They have their own place in memory, where they don't get wiped out every time the function is called.[4]

[3] The memory addresses are written as hexadecimal (base-16) numbers.

[4] The non-static variables we've been using so far are formally called "automatic" variables. If we wanted to, we could explicitly use the word "auto" in front of the variable definition to show this, but that's seldom done.

Program 9.11: counter.cpp

```
#include <stdio.h>

void myfunc () {
  static int count = 0;
  if ( count == 0 ) {
    printf ( "This is the first time we've used this function\n" );
  } else {
    printf ( "We've already used this function %d times\n", count );
  }
  count++;
}

int main () {
  int i;
  for ( i=0; i<5; i++ ) {
    myfunc();
  }
}
```

Notice that we can still intialize a `static` variable. In Program 9.11 we set the initial value of `count` to zero. This is only done once, the first time the function is used. The variable won't be reset to zero every time we call the function.

If we ran this program, we'd see something like this:

```
This is the first time we've used this function
We've already used this function 1 times
We've already used this function 2 times
We've already used this function 3 times
We've already used this function 4 times
```

If we had omitted the word "`static`", the variable `count` would be wiped out and reset to zero every time we used the function, so it would just keep repeating "This is the first time we've used this function".

9.11. Passing Addresses

What if we really want one function to be able to change the value of a variable in another function? To do that, we need to know where to find the variable in the computer's memory. As we've seen before, we can use an & in front of a variable's name to get its memory address. But how do we tell the computer to stick a value into a particular memory address? C provides another symbol, "*" that we can use to help us do this.

Program 9.12 is a modified version of Program 9.9. In the new version, instead of giving changenum the *value* of mynum, we give it the *address* of mynum.

Looking for the right (memory) address?

Source: Wikimedia Commons

Program 9.12: passing.cpp

```
#include <stdio.h>

void changenum ( int *number ) {
  printf ( "Multiplying %d by 1000...\n", *number );
   *number =  *number * 1000;
  printf ( "...the result is %d\n", *number );
}

int main () {
  int mynum = 1234;
  changenum( &mynum );
  printf ( "My number is now %d\n", mynum );
}
```

Memory addresses can be stored in special variables called "pointers". In our new definition of changenum, we say that this function should get a "pointer to an integer" (int *) as its argument. Pointers "point" at the memory location where some data is stored. The * means that this variable is a pointer.

Inside changenum we use the * operator in another way. The expression "*number = ..." means "set the variable at this memory location to ..."[5].

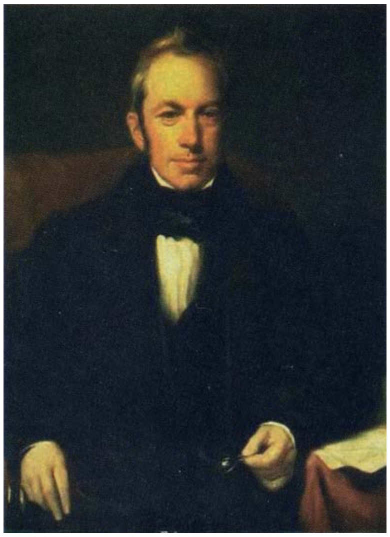
If we ran Program 9.12 we'd see this, showing that we have actually changed the value of mynum:

```
Multiplying 1234 by 1000...
...the result is 1234000
My number is now 1234000
```

[5] Programmers call & the "referencing operator" and * the "dereferencing operator".

9.12. Bouncing Molecules

In 1827, botanist Robert Brown noticed something odd while looking at pollen grains in water, through a microscope. The pollen grains were very small, but he saw that they emitted even smaller particles that we now know were bits containing starch and fat. This in itself was interesting, but Brown was also fascinated by the fact that these tiny particles moved around continously, as though they were alive.



Robert Brown, botanist, by Henry William Pickersgill (1782-1875). For much more information about Brown's work, see this modern-day recreation of it by researchers at Hamilton College.

Source: Wikimedia Commons

On further experiments with inorganic matter like bits of glass and granite, he found that those particles displayed the same behavior. What caused them to move? Today we know that the particles Brown observed were being jostled by water molecules, and we call this phenomenon "Brownian Motion".

Program 9.13 simulates the motion of a tiny particle floating on the surface of some water. It begins by picking a random starting position for the particle by setting the *x* and *y* coordinates of the particle's position to random numbers between zero and one.

The program then tracks the particle through 10,000 collisions. Each collision moves the particle by some random amount. The function `move` takes the particle's current *x* and *y* coordinates and changes them to new values by adding a random amount between -0.5 and 0.5. Notice that we give `move` the *addresses* of `x` and `y`, making it possible for the function to change the values of these variables, as described in Section 9.11 above.

The program prints each new position, so we could plot the particle's path if we wanted to. Figure 9.16 shows the path of a typical particle.

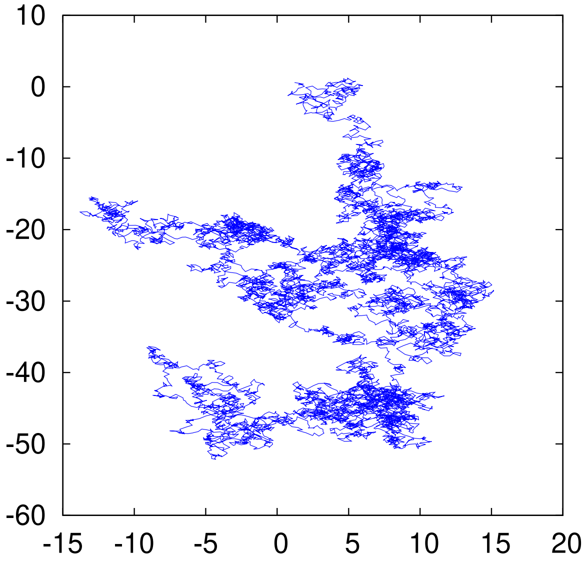


Figure 9.16: The path of a typical particle in our brownian motion simulation. For a deeper investigation of the mathematics of random walks, see this video from the PBS show "Infinite Series": https://www.youtube.com/watch?v=stgYW6M5o4k

Program 9.13: brownian.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void move ( double *x, double*y ) {
  *x = *x + rand()/(1.0+RAND_MAX) - 0.5;
  *y = *y + rand()/(1.0+RAND_MAX) - 0.5;
}

int main () {
  double x, y;
  int i;

  srand(time(NULL));

  // Pick a random initial position:
  x = rand()/(1.0+RAND_MAX);
  y = rand()/(1.0+RAND_MAX);

  // Move around:
  for ( i=0; i<10000; i++ ) {
    move( &x, &y );
    printf ( "%lf %lf\n", x, y );
  }
}
```

9.13. Passing Arrays to Functions

Source: Wikimedia Commons

Sometimes we want a function to operate on an array. We can do that, as shown in Program 9.14, which finds the biggest element of an array of doubles. We could use this to find the most heavily-laden coal car in our coal train example from Chapter 6, for example.

Program 9.14 fills an array with random values, and then uses the function maxelement to find the element that contains the largest value.

Program 9.14: findmax.cpp

```
#include <stdio.h>
#include <stdlib.h>

int maxelement ( int size, double array[] ) {
  double max;
  int imax;
  int i;
  for ( i=0; i<size; i++ ) {
    if ( i == 0 ) {
      max = array[i];  // Use first number as first guess.
    } else {
      if ( array[i] > max ) {
        max = array[i];
        imax = i;
      }
    }
  }
  return ( imax );
}

int main () {
  double array[100];
  int i;
  for ( i=0; i<100; i++) {
    array[i] = rand();
    printf ( "%d %lf\n", i, array[i] );
  }

  printf ( "The biggest element is number %d\n",
           maxelement( 100, array ) );
}
```

Notice that we tell the compiler that one of maxelement's arguments will be an array by putting [] after the variable name. Also notice that we need to tell the function how big the array is. The size argument to maxelement tells the function how many elements are in the array.

Exercise 49: Dot Products

Imagine you have two 3-dimensional vectors, **A** and **B**. Each of these can be represented as 3-element array in C. In mathematics, the "dot product" of two 3-d vectors is

$$\mathbf{A} \cdot \mathbf{B} = \sum_i A_i B_i \tag{9.5}$$

or, writing out the sum:

$$\mathbf{A} \cdot \mathbf{B} = A_0 B_0 + A_1 B_1 + A_2 B_2 \tag{9.6}$$

Write a function that takes two 3-element `double` arrays as arguments and returns their dot product as a `double` value.

Test your function with a program that multiplies these two arrays together and prints out their dot product:

```
double a[3] = { 1.0, 2.0, 3.0 };
double b[3] = { 4.0, 5.0, 6.0 };
```

9.14. The Chaos Game

Let's write another program that passes arrays to a function. This program will play "The Chaos Game". the rules of this game are:

In Discordianism, Eris is regarded as the goddess of chaos. She says "I am the substance from which your artists and scientists build rhythms. I am the spirit with which your children and clowns laugh in happy anarchy. I am chaos. I am alive, and I tell you that you are free." (from the *Principia Discordia*).

Source: Wikimedia Commons

1. Pick three reference points on a piece of paper and label them ***P*1**, ***P*2**, and ***P*3**.
2. Draw another point (let's call it ***b***) anywhere on the paper.
3. Randomly choose one of the reference points. For example, you could roll a 6-sided die and pick ***P*1** if you get **1** or **2**, ***P*2** if you get **3** or **4**, and ***P*3** if you get **5** or **6**.
4. Draw a new point halfway between ***b*** and the reference point you picked. This point becomes the new ***b***.
5. Go back to step 3 and repeat.

Doing this by hand would get boring pretty quickly, so let's write a computer program to do it for us. Program 9.15 repeats steps 3 and 4 10,000 times, printing the new coordinates of ***b*** each time.

At the top of the program we define our three reference points. Each point is represented by a 2-element array containing the point's x and y coordinates. The function `movehalf` takes the point $\boldsymbol{b}$ (also represented by a 2-element array) and moves it half of the way toward one of our reference points.

Notice that we give $\boldsymbol{b}$ to `movehalf` as an array. You might remember that, in Chapter 8, we learned that C programs interpret an array's name (without an element number after it) as the *memory address* of the array. This means that when we give a function an array as one of its arguments, the function is able to change the values of the array elements, just as when we give a function the memory address of a single variable (which we saw above, in Section 9.11).

The `main` function picks random starting values for the coordinates of our point $\boldsymbol{b}$, then uses the rules of the Chaos Game to move this point around. Instead of rolling a die, the program generates a random number between zero and one. If this number is less than **1/3** the program moves the point halfway to point $\boldsymbol{P}$**1**. If it's between **1/3** and **2/3** it moves toward $\boldsymbol{P}$**2**. If it's between **2/3** and **1**, it goes toward $\boldsymbol{P}$**3**.

We could save the program's output in a file by typing "`./chaos > chaos.dat`", and then we could graph these points with the *gnuplot* command "`plot "chaos.dat" with dots`". The result is shown in Figure 9.17.

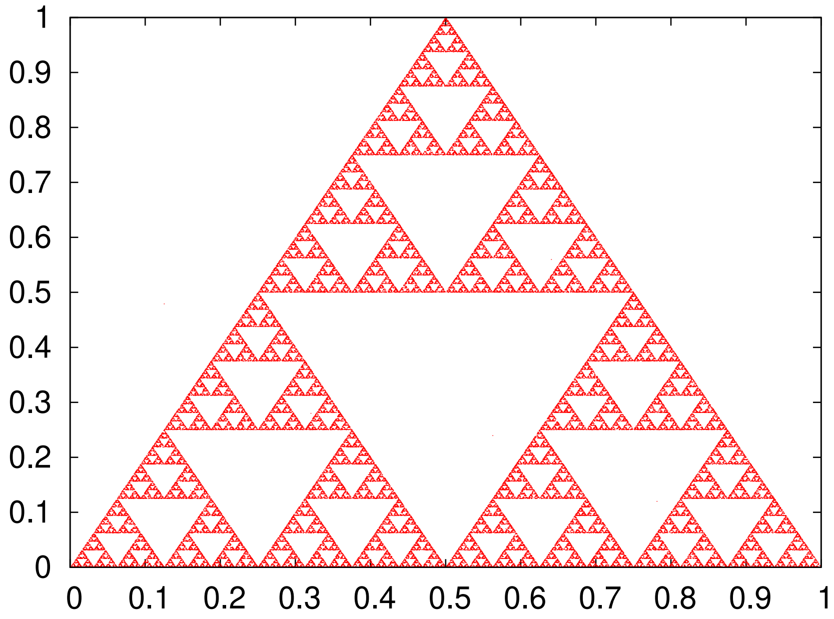


Figure 9.17: A "Sierpinski Triangle" (or "Sierpinski Gasket") drawn by playing The Chaos Game. Telling *gnuplot* to use "dots" causes it to draw small points instead of symbols. For more on The Chaos Game see this Numberphile video: https://www.youtube.com/watch?v=kbKtFN71Lfs

You might well be surpised by this result! The shape you see is called a Sierpinski Triangle (or Sierpinski Gasket). Since we picked a random direction each time, you might have expected the points to be spread evenly around the page. In fact, no matter where you start on the page, the points will eventually be “attracted” to the red areas on the graph. This shape is an example of a “chaotic attractor”. Even though we can’t predict where a given point will land on the graph, the overall pattern of all the points is very orderly and well-defined. This is an example of order emerging spontaneously from randomness. Such phenomena are common in the natural world, where simple underlying rules can lead to intricately beautiful structures.

Program 9.15: chaos.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

double p1[2] = {0,0};
double p2[2] = {0.5,1.0};
double p3[2] = {1.0,0.0};

void movehalf ( double b[], double point[] ) {
  b[0] = b[0] + 0.5*( point[0] - b[0] );
  b[1] = b[1] + 0.5*( point[1] - b[1] );
}

int main () {
  double r, b[2];
  int i;

  srand(time(NULL));

  b[0] = rand()/(1.0+RAND_MAX);
  b[1] = rand()/(1.0+RAND_MAX);

  for ( i=0; i<10000; i++ ) {

    r = rand()/(1.0+RAND_MAX);
    if ( r < 1.0/3 ) {
      movehalf( b, p1 );
    } else if ( r < 2.0/3 ) {
      movehalf( b, p2 );
    } else {
      movehalf( b, p3 );
    }

    printf ( "%lf %lf\n", b[0], b[1] );
  }
}
```

9.15. Command-Line Arguments

If `main` is just a function, can we give it arguments? Yes we can, but they must always be a particular pair of arguments. It turns out that the arguments given to `main` contain anything you type on the command line after the name of your program. These extra things are called "command-line arguments".

Take a look at Program 9.16. This program uses several new concepts. First, notice that `main` now has two arguments, `int argc` and `char *argv[]`. The `main` function must always have either these two arguments or none at all. The first argument, `argc`, tells the program how many arguments you typed on the command line when you ran the program. The second argument is an array of character strings, each element of which contains one of the command-line arguments.

Program 9.16: args.cpp

```
#include <stdio.h>
int main ( int argc, char *argv[] ) {
  int i;

  for ( i=0; i<argc; i++ ) {
    printf ( "argv[%d] = \"%s\"\n", i, argv[i] );
  }
}
```

Let's see what happens when we run the program. If we just type `./args` the program says:

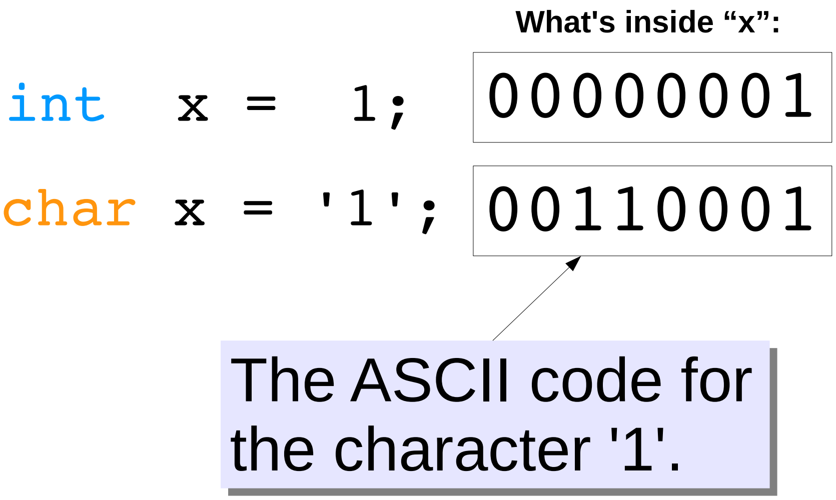
```
argv[0] = "./args"
```

`argv[0]` will always contain the name of the program itself, as it's typed on the command line. Now look what happens if we type `./args hello 1 2 3` on the command line:

```
argv[0] = "./args"
argv[1] = "hello"
argv[2] = "1"
argv[3] = "2"
argv[4] = "3"
```

We could use these command-line arguments to control our program's behavior if we wanted to.

Notice, however, that all of the elements of `argv` are character strings, not numbers. In the example above, `argv[1]` is equal to the *character string* `"1"`, not the number 1. We can see how these differ by looking at how each value is stored in the computer's memory:



```
int  x =  1;
char x = '1';
```

C's standard libraries provide a pair of functions for converting strings to numbers: `atof` and `atoi`. The `atof` function converts a character string into a floating-point number (a `double`), and `atoi` converts a string into an `int`. In order to use these functions, we need to add `#include <stdlib.h>` at the top of the program. In the next section we'll look at a program that uses these functions.

Here's a simple program that illustrates the use of `atoi` to convert a command-line argument into an `int`. The program counts up to a number given on the command line. For example, if you said:

```
./countto 10
```

the program would count to ten.

Program 9.17: countto.cpp

```
#include <stdio.h>
#include <stdlib.h>
int main ( int argc, char *argv[] ) {
  int i,n;
  n = atoi( argv[1] );
  for ( i=0; i<=n; i++ ) {
    printf ( "%d\n", i );
  }
}
```

Convert argument to `int`

9.16. Command-Line Cannon

Let's use command-line arguments to write an improved version of our earlier "cannon" program. Program 9.18 shows a modified version of Program 9.8, omitting the definitions for functions other than main. (The other functions will be the same for both programs.)

The new version of the program lets us enter the angle and intial velocity on the command line when we run the program, instead of asking the user for these values. For example, we could type:

```
./cannon 45 27
```

to point the cannon at a 45°angle and specify an initial velocity of 27 m/s.

Program 9.18 first checks to see if you've given it the right number of command-line arguments by looking at the value of argc. We want to make sure the user has given values for theta and vinit. If not, the program prints out a friendly usage message and stops the program.

We can stop the program at any time by using the "exit" function, which is part of C's standard library of functions. exit takes one argument: an integer number specifying the exit status of the program. This can be any number you like, but usually anything other than zero means that the program failed. You can put an "exit(0);" statement at the end of your programs, but it's not necessary.

Notice that we check to make sure argc is equal to 3. Why 3? Won't there be only two arguments, theta and vinit? The argv array actually contains one extra thing: the name of the program itself. If we type "./cannon 45 27", the elements of argv look like this:

```
argv[0] = "./cannon";
argv[1] = "45";
argv[2] = "27";
```

Program 9.18 uses atof to convert the command-line values of theta and vinit into numbers.

Finally, program 9.18 uses the program name as part of the friendly error message it prints if the user doesn't supply enough command-line arguments.

Program 9.18: cannon.cpp, with command-line arguments

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

// Other functions go here....

int main ( int argc, char *argv[] ) {
  double vinit;
  double theta;

  if ( argc != 3 ) {
    printf ( "Syntax: %s theta vinit\n", argv[0] );
    exit(1);
  }

  theta = atof( argv[1] );
  theta = to_radians( theta );

  vinit = atof( argv[2] );

  printf ( "Time of flight is %lf sec.\n",
           time_of_flight( vinit, theta ) );

  printf ( "Max height is %lf meters.\n",
           max_height( vinit, theta ) );

  printf ( "Range is %lf meters.\n",
           range( vinit, theta ) );
}
```

Exercise 50: Hang Time

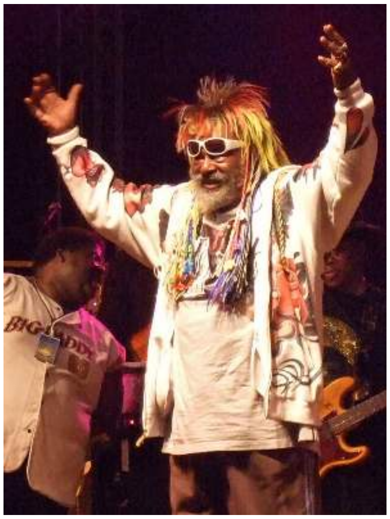
Write a program like Program 9.18 using the `time_of_flight` function from Program 9.8. The program should accept two command-line arguments, `theta` and `vinit`, and it should print the time of flight based on the values supplied by the user.

Keeping the angle at 45°, run your program repeatedly to find the minimum initial velocity (to the nearest m/s) the acrobat would need if she wanted to remain in the air for at least 25 seconds (matching a ride on the Vomit Comet).

9.17. Passing Functions to Other Functions

We've written a lot of programs that print a list of x and y values. The "Red Baron" program (Program 9.6) earlier in this chapter is a recent example. These programs loop through a bunch of x values, compute the y value for each, and print the results. The value of y is given by some function of x. The function might be something simple like `sqrt(x)` or it might be something complicated, like the Red Baron's flight path.

No matter what the function is, though, we often do the same thing with it: calculate its value for several x values and print the result. Wouldn't it be nice if we had a function that would accept the name of a "y-generating" function, and print a list of x and y values using it? Let's make one!



George Clinton, "the Godfather of Funk".

Source: Wikimedia Commons

Take a look at Program 9.19. As you can see, its `main` just contains one statement. This statement uses the function `plotit` to produce a list of 100 x and y values for $y = \sqrt{x}$, with values of x ranging between zero and 500. The first argument to `plotit` is the *address* of the `sqrt` function. Just as we passed the addresses of variables to a function in Section 9.11, we can also pass the address of a function.

At the top of the program is the `plotit` function. To tell `plotit` to expect a function address, we write one of its arguments as:

```
double (*func)(double)
```

which means "this argument will be the address of a function that takes a `double` as its only argument and returns a `double`". This kind of argment is called a "function pointer" because it points to the memory address of a function. In general, such an argument will have this form:

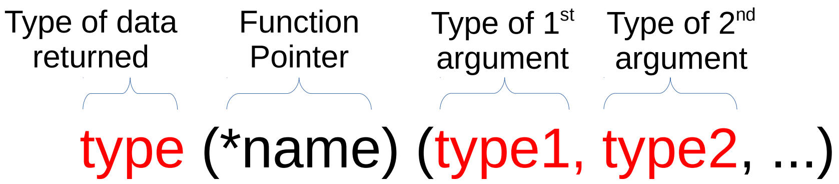


Figure 9.18: General form of a function pointer.

Inside plotit, we can use the name func to refer to the function, no matter what it really is.

When we use the plotit function, we give it the address of the function to be plotted by writing the function's name, just as we do for arrays.

We could replace sqrt with cos to get a list of values for $y = cos(x)$, or we could use exp to get $y = e^x$. We can use any function that takes a double as its only argument and returns a double.

Program 9.19: funcplot.cpp

```
#include <stdio.h>
#include <math.h>

void plotit ( double (*func)(double), int nsteps, double xmin, double xmax ) {
  int i;
  double x, step;
  step = (xmax - xmin)/nsteps;

  x = xmin;
  for ( i=0; i<nsteps; i++ ) {
    printf ( "%.10e %.10e\n", x, func(x) );
    x += step;
  }
}

int main () {
  plotit( sqrt, 100, 0, 500 );
}
```

Finally, notice that plotit uses the format %.10e when printing numbers. As we've seen before, the .10 tells printf to print ten decimal places. The plotit function uses e instead of lf to tell printf to print the numbers in scientific notation. This gives our function the ability to print a wide range of numbers.

9.18. Using qsort for Sorting

In Chapter 6 we looked at the "Bubble Sort" algorithm, which we used for sorting the elements of an array. Now let's look at a faster, more flexible way of sorting array elements.

Licorice Allsorts are among the author's favorite candies. Yum!

Source: Wikimedia Commons

The C standard libraries contain a function named qsort (for "Quick Sort") that can be used to sort any kind of array. To use it, we give qsort the name of the array to be sorted, and the address of a function (written by us) for comparing any two array elements. When the function compares two elements (let's call them element *a* and element *b*) it should return zero if the two elements are the same, -1 if *a* is less than *b*, or 1 if *a* is greater than *b*.

Since qsort can be used to sort any type of array, we can't assume that the array elements will be ints or doubles or any other specific type. Because of this, qsort passes *a* and *b* to our comparison function as void variables (variables without any specific type), and it's up to the comparison function to figure out how to use them.

Here's what a comparison function for comparing two integers might look like:

```
int compare_int(const void *i1, const void *i2){
  int a, b;

  a = *(int *)i1;
  b = *(int *)i2;

  if ( a<b ) {
    return (-1);
  } else if ( a>b ) {
    return (1);
  } else {
    return (0);
  }
}
```

As you can see, the two arguments given to the comparison function are the addresses (notice the asterisks) of two void variables (meaning variables of any type). The function first needs to convert those into integers. It does this by converting the void addresses into int addresses, then it puts another asterisk on the left to get the actual integer values stored at those addresses and store them in the variables a and b. Then it's just a straightforward "if" statement to compare the two numbers and return zero, -1, or 1, whichever is appropriate.

Program 9.20 reads an unsorted list of integers from a file and uses the qsort function to sort them. The unsorted numbers are in a file named unsorted.dat. The program will read as many numbers as are in the file, up to a maximum of 1,000 numbers. After the numbers are sorted, the program prints them in their sorted order, from smallest to largest.

Program 9.20: sortit.cpp

```
#include <stdio.h>
#include <stdlib.h>
int compare_int(const void *i1, const void *i2){
  int a, b;

  a = *(int *)i1;
  b = *(int *)i2;

  if ( a<b ) {
    return (-1);
  } else if ( a>b ) {
    return (1);
  } else {
    return (0);
  }
}

int main ( int argc, char *argv[] ) {
  FILE *input;
  const int nmax = 1000;
  int i, n=0, numbers[nmax];

  input = fopen( "unsorted.dat", "r" );
  while ( n<nmax &&
          fscanf( input, "%d", &numbers[n] ) == 1 ) {
    n++;
  }
  fclose ( input );

  qsort( (void *)numbers, n, sizeof(int), compare_int );

  for ( i=0; i<n; i++ ) {
    printf( "%d\n", numbers[i] );
  }
}
```

Program 9.20 gives the qsort function four arguments:

1. The name of the array to be sorted, cast as a "(void *)". This gives qsort the memory address of the array, without specifying any particular variable type.
2. The number of elements to be sorted. Note that this doesn't have to be all of the elements in the array. In the example above, if unsorted.dat only contained 100 numbers, then n would be 100, even though the array has $1,000$ elements.
3. The size (in bytes) of each element of the array. Since qsort doesn't know what kind of elements it's sorting, we need to tell it how big they are. Here we use the sizeof statement that we introduced in Chapter 6.
4. Finally, we give qsort the name of our comparison function. In this case, it's just the compare_int function we wrote above.

Coat of arms of "Sort", a town in Catalonia, Northwest Spain. Its name means "luck" in the Catalan language.
Source: Wikimedia Commons

When our comparison function compares two elements, we're free to define what we mean by "greater than", "less than", or "equal". Why would we need this flexibility? Imagine, for example, that we had a coal train with many cars, each with a different amount of coal, and each destined for a different customer. We might have an array of customer IDs. We could just sort the array in order of increasing ID number, but we might sometimes want to sort the list of IDs based on how much coal they ordered, or by how many miles it is from the coal mine to the customer. The qsort function gives us the flexiblity to do that[6].

[6] Later on, in Chapter 12, we'll see that C lets us define our own, complicated, variable types. The qsort can even be used with those, since we get to define our own comparison function.

9.19. Conclusion

Writing your own functions in C is easy, and can be beneficial in several ways. Using functions can help you:

- Avoid duplicating the same code many times within a program.

  If you find yourself typing the same set of statements again and again, it's time to think about creating a function to replace them.

- Make your program easier to modify.

  After you've encapsulated a task within a function, you can easily modify it to make it better, without having to modify the rest of your program.

- Re-use your code in other programs.

  Once you've written your function, you can re-use it in other programs.

- Catch programming mistakes.

  The compiler makes some syntax checks when a function is called, so this is an opportunity to catch mistakes.

- Avoid accidentally changing variables.

  As we've seen, variables inside a function are independent from variables of the same name in other functions.

I encourage you to get into the habit of writing code that breaks work up into bite-sized functional chunks. Modularizing your programming jobs keeps you from reinventing solutions, and helps unclutter the visual flow of your programs, making it easier to see what the program is doing.

Later, we'll be learning how to create your own libraries of pre-compiled functions that you can reuse again and again.

Practice Problems

1. As we saw in Chapter 7, a Normal (or Gaussian) curve is described by the equation:

   $$P(x) = Ae^{-\frac{(x-\overline{x})^2}{2s^2}}$$

   If we let $A = 1$, $\overline{x} = 10$, and $s = 1$ the equation gets simplified to:

   $$P(x) = e^{-\frac{(x-10)^2}{2}}$$

   Write a program named `gauss.cpp` that contains a function named `P` that returns the value of $P(x)$ from the simplified equation above. Note that you'll need to include `math.h` at the top of your program so you can use the `exp` and `pow` functions. The function should take one `double` argument (the value of x) and return a `double` value.

   In the `main` part of your program, create a loop that steps through 200 values of x, from zero to 19.9 in steps of 0.1. For each value of x, print x and $P(x)$.

   If you put the program's output into a file and plot it with *gnuplot* you should see something like Figure 9.19.

2. Write your own version of Program 9.17 (`countto.cpp`) that adds a check to make sure the user has supplied a number on the command line. If the user doesn't give a number, the program should print a friendly message describing how to run the program, then exit without doing anything else. See Program 9.18 for an example that shows how to use the `exit` function.

3. Write a program named `sphere.cpp` that calculates the volume of a sphere, given its radius. Remember that the formula for the volume of a sphere is $V = \frac{4}{3}\pi r^3$. Use command-line arguments, as described in Section 9.15 above, to allow the user to specify the radius on the command line. For example, for a sphere of radius 3.5, the user should be able to run the program like this:

   ```
   ./sphere 3.5
   ```

   The program should just print out the calculated volume with no commentary. So, if the sphere's radius is 3.5, the program should print `179.594380`.

   Make sure the program checks to see if the user has specified the radius, and print an error message and exit if they haven't. See Program 9.18 for an example of this.

   Hints: Since the radius can, in general, contain decimal places you'll need to use `atof` to convert the command-line argument to a number. See Program 9.18 for an example.

"Functional" is a solo piano piece composed by Thelonious Monk, "the genius of modern music".

Source: Wikimedia Commons

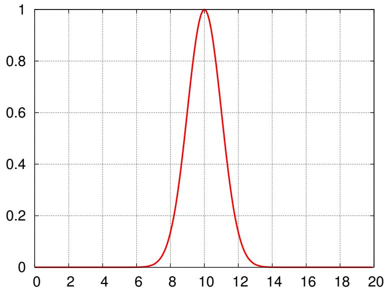


Figure 9.19: Your `gauss.cpp` program's output should look like this if you plot it with *gnuplot*.

The *Trylon* and *Perisphere* were two buildings made for the 1939 New York World's Fair. The Perisphere had a diameter of 180 feet. You could travel through it on a moving sidewalk and look down a diorama depicting a utopian city.

Source: Wikimedia Commons

4. Write a program named `compare.cpp` that contains a C function named "`similar`" that compares two `double` numbers and returns an integer value of 1 if the numbers differ by less than 0.0001, or 0 if they're farther apart. The program should ask the user for the two numbers to compare, and read them in with `scanf`. The program should use your function to compare the numbers, and tell the user (in clear, friendly words) if they're within 0.0001 of each other.

5. The formula for computing the amount of money in a savings account is:

$$M_{now} = M_{orig}\left(1+\frac{r}{n}\right)^{nt}$$

   where M_{now} is the amount of money you currently have, M_{orig} is the amount you originally deposited, r is the interest rate the bank is paying you, n is the number of times per year that the interest is added to your account, and t is the number of years since you originally deposited the money.

   Write a program named `lucre.cpp` containing a function named `mnow` that begins like this:

```
double mnow ( double morig, double rate, int ntimes, int years )
```

   where the arguments correspond to M_{orig}, r, n, and t, in that order. the function should use these arguments to compute M_{now}.

   Your program should ask the user for the original amount of money in her/his account, then print how much money will be in the account each year for the next 20 years, assuming an interest rate of 0.05 (5%) with interest added 4 times per year. Use your `mnow` function to do the calculations. The program's output should be two columns: year number (0, 1, 2, ...*etc.*) and M_{now} for that year.

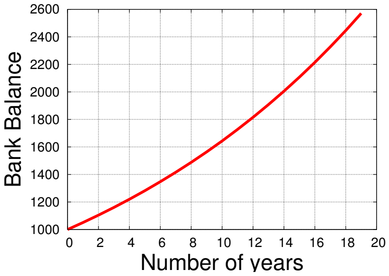


Figure 9.20: Bank balance over 20 years, starting with $1,000, with 5% interest compounded 4 times per year.

6. Write a program named `volumes.cpp` that calculates the volumes of some common shapes. Define three functions named `vsphere`, `vbox`, and `vcone` to calculate the volume of a sphere, a box, and a cone, respectively. Your definition of the vsphere function should start like this:

```
double vsphere ( double r )
```

   where `r` is the sphere's radius. The definition of `vbox` should start like this:

```
double vbox ( double l, double w, double h )
```

   where `l`, `w`, and `h` are the length, width, and height of the box. The definition of `vcone` should start like this:

```
double vcone ( double r, double h )
```

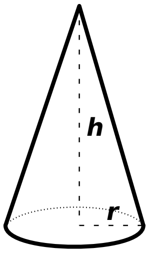


Figure 9.21: The volume of a **cone** is $\frac{1}{3}\pi r^2 h$, where r and h are as shown in the diagram above.
The volume of a **sphere** is $\frac{4}{3}\pi r^3$.
The volume of a **box** is just length × width × height.

where r is the radius of the cone's base and h is the cone's height. See Figure 9.21 for the formulas you'll need.

Your program should ask the user to specify which shape to use like this:

```
Enter the type of shape (1=sphere, 2=box, 3=cone):
```

then the program should ask the user for the dimensions of the shape, calculate its volume using the appropriate function, and tell the user the result.

7. Create a program that adds two numbers: Write a program named `add.cpp` that accepts two integers as command-line arguments (see Section 9.15 above). The program should add the two numbers and tell you what their sum is. For example, if you type this:

   ```
   ./add 23 52
   ```

   The program should print "75".

8. Write a program named `maxnum.cpp` that accepts a list of numbers on the command line and tells you which of the numbers is the largest. The program should accept numbers with decimal places, so you'll need to use `double` variables and the `atof` function. (See Section 9.15 above for information about using command-line arguments.) You should be able to run your program like this, for example:

   ```
   ./maxnum 12 13 128 765 2 4 3 -78
   Maximum number is 765.000000.
   ```

   Make sure your program can deal properly with negative numbers. If it's given the numbers -2 and -5, it should tell you that the largest number is -2.

9. Write a program named `testprime.cpp` that checks to see if a given integer is prime. (Remember that a prime number is one that can only be divided evenly by itself and 1.) The program should accept the number to be tested on the command line. For example:

   ```
   ./testprime 8675309
   ```

   The program should say something like "8675309 is prime" or "8675309 is NOT prime". The program should check the value of `argc` to make sure the user has supplied a number to be checked. If not, the program should tell the user what to do, and use `exit(1)` to stop. See Section 9.15 above for information about using command-line arguments.

   If we call the number to be checked n, then your program should look to see if n can be divided by any of the numbers from 2 to

Speaking of adders, the harmless Eastern Hognosed Snake (*Heterodon platyrhinos*) is sometimes called a "puff adder" because it tries to frighten you by spreading its head like a cobra and hissing. If that doesn't work, it will roll over onto its back and play dead. If you turn it upright, it will roll over again just to prove that it's really dead. ("Don't bother me! I'm busy being dead!")

Tommy Tutone, the band responsible for the 1981 hit song *867-5309/Jenny*.

Source: Wikimedia Commons

n-1, inclusive. You can use the % operator to check each number. Remember that n%i will be zero if n can be evenly divided by i. Refer to Chapter 4 for more information about the % or "modulo" operator.

Note that your program will only be able to work on numbers that are small enough to fit into an integer variable. On most computers, the biggest number that can fit into an int will be 2,147,483,647.

10. Write a new version of Program 9.6 (redbaron.cpp) that uses a different function for the flight path. Instead of the complicated function in Program 9.6, use:

$$h(x) = 10000 - \frac{(x - 3000)^2}{10000}$$

and modify the "for" loop so that it does 6,000 steps instead of 1,000, tracking the plane over a distance of 6,000 meters.

Run your program and redirect the output into a file, then plot the file using *gnuplot*. What shape does it make? The graph should approximate the path followed by a "zero-G" aircraft near the top of its trajectory (see Figure 9.7).

Teacher and astronaut Christa Macauliffe experiencing weightlessness in NASA's "Vomit Comet". She died tragically in 1989, when the space shuttle *Challenger* exploded shortly after launch.

Source: Wikimedia Commons

11. In physics and math we often want to go through a list of numbers "cyclically". By this I mean that when we get to the end of the list we start back at the beginning again. For example, if our list contained the numbers $1, 2, 3$ we could start at any of the numbers and write them down, in order, starting back at the beginning if necessary, until we'd written them all.

We could write this cyclic list in any of the following equivalent ways:

```
1 2 3
2 3 1
3 1 2
```

Notice that the list rotates in a particular direction, clockwise in this case, as shown in Figure 9.22.

Imagine that we have three variables, i, j, and k with initial values i=1, j=2, and k=3. Write a function named rotate that uses the techniques described in Section 9.11 above to change the values of these variables, moving the value of i to j, the value of j to k, and the value of k to i. The function should start out like this:

```
void rotate ( int *i, int *j, int *k )
```

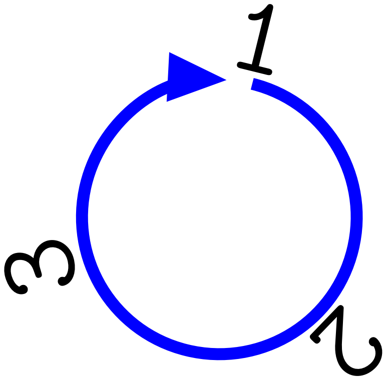


Figure 9.22: A cyclic list of numbers.

Use your function in a program named cycle.cpp that prints out the initial values of i, j, and k, then uses your rotate function to "rotate" the values of the three variables three times, printing out their new values after each rotation.

12. Sometimes a program needs to accept a file name on the command line. Write a program named `randfile.cpp` that can be run like this:

```
./randfile random.dat
```

When the program is run like this, it should generate 1,000 random numbers and write them into a new file named `random.dat`. You can just use the `rand` function to generate each random number.

The program should check to make sure the user has supplied a file name, and print an error message and exit if not. (See Program 9.18 for an example of this.)

Hint: The command-line arguments `argv[1]`, etc., are character strings, so you don't need to do any conversion with `atoi` or `atof`. In this program, you can just put `argv[1]` in place of the file name in your `fopen` statement.

The fascinating properties of strings.(*Sitzendes Mädchen mit einer Katze*, 1903, by Albert Anker.)

Source: WikiArt

13. A character string is just an array of characters. As we saw in Chapter 8, C provides us with a handy `strlen` function that can tell us the length of the text stored inside a character string. The `strlen` function does this by looking for the special "NUL" character that terminates the string.

Complete the following program (named `fakestrlen.cpp`) by adding a function named `mystrlen` that does the same thing the built-in `strlen` function does.

```
#include <stdio.h>
int mystrlen ( char string[] ) {
   // Insert your function here.
}
int main () {
  char string1[] = "Help, I'm trapped in a computer!";
  char string2[] = "Just kidding!";
  char string3[] = "They made me say that!";

  printf ( "String 1 length is %d\n", mystrlen( string1 ) );
  printf ( "String 2 length is %d\n", mystrlen( string2 ) );
  printf ( "String 3 length is %d\n", mystrlen( string3 ) );
}
```

Hints: Your function should use a `while` loop that starts with the first character of the string (character number zero) and checks each character to see if it's the NUL character, which is written as '`\0`' in C. The loop should continue for as long as the current character isn't a NUL. When the loop is done, the function should return the number of the current character.

14. Hot objects tend to emit heat and light in a range of wavelengths. The temperature of the object determines which wavelengths are emitted the most. In 1900 Max Planck wrote down the modern mathematical description of these emissions (known as "black body radiation"). The relationship between the intensity, I of radiation at a given wavelength, λ, depends on temperature, T, like this:

$$I(\lambda) = \frac{2hc^2}{\lambda^5} \frac{1}{\exp(\frac{hc}{\lambda kT}) - 1}$$

where **exp** is the exponential function and the physical constants are (in SI units):

Symbol	Name	Value
The speed of light in a vacuum	c	2.99792458×10^{8}
Planck's constant	h	$6.62606896 \times 10^{-34}$
Boltzmann's constant	k	$1.3806504 \times 10^{-23}$

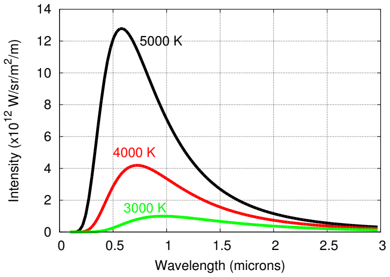


Figure 9.23: $I(\lambda)$ for several temperatures. Notice that the peak of I moves to the left as temperature increases. This shows that hotter objects emit more high-frequency radiation. (Low wavelengths correspond with high frequencies, and vice versa.)

Write a function named `intensity` that begins like this:

```
double intensity ( double lambda )
```

where `lambda` is λ and the function returns the value of $I(\lambda)$ from the equation above. Use a global variable named `t` to set the temperature to 5,000 Kelvin.

Use this function in a program named `planck.cpp`. The program should also contain the function named `plotit` that we used in Section 9.17. Have your program use the `plotit` function to print 100 values of $I(\lambda)$, with λ going from `0.1e-6` meters to `3e-6` meters. If you plot your results with *gnuplot* you should see a curve like the largest curve in Figure 9.23.

15. Using the technique shown in Program 2.4 (diceroll.cpp) in Chapter 2, write a program that emulates a die with an abritrary number of sides. Call the new program `unidie.cpp`. The user should be able to specify the minimum and maximum numbers on the die by giving the program command-line arguments. The program should contain a function named `roll` that takes `min` and `max` as arguments and returns a random integer between `min` and `max`, inclusive. Make sure the program checks to see if the user has provided the necessary command-line arguments, and takes apprpriate action if not. When the program is run, it should print out the random number "rolled" by the die.

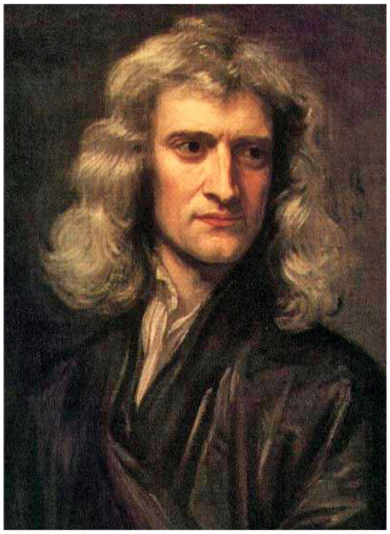
Detail from *Vanitas* by Adriaan Coorte.

Source: Wikimedia Commons

10. Numerical Integration

10.1. Introduction

Sometimes a problem in Science or Engineering allows us to find an elegant solution that represents a simple, exact answer. Mathematics tells us that the area of a circle is exactly πr^2. We know that the distance travelled by a uniformly accelerating body is $v_0 t + \frac{1}{2}at^2$.



Isaac Newton in 1689.

Source: Wikimedia Commons

Calculus (first called the "calculus of infinitesimals") was co-invented in the 17th Century by Gottfried Wilhelm von Leibniz in Germany and Isaac Newton in England. The two argued bitterly over which of them deserved credit. The Royal Society of London formed a committee chaired by Newton to investigate the dispute, and its report (written by Newton) ruled, unsurprisingly, in favor of Newton. Today mathematicians give both men equal credit.

Elegant answers aren't always available, though. Often, a simple mathematical solution eludes us. This can happen because of some inherent feature of the problem that makes it mathematically difficult, or because the sheer size of the problem makes it intractible, or because we only have a little bit of data.

For problems like this, we need to apply brute force. We chip away at the problem, hoping to find an approximation that's good enough to satisfy our immediate needs. Fortunately, we often find that, by working hard enough, we can make our approximation as close to the true answer as we like.

One place this kind of problem crops up is in the evaluation of integrals. As you know if you've taken Calculus, the evaluation of integrals can be difficult. Much of what you learn in Calculus class consists of tricks for evaluating various kinds of integrals.

An integral is conceptually simple, though: it's just adding things up. Mathematician Gottfried Leibniz introduced the integral sign, $\int$, which is a stretched-out "S", for "Sum".

Since computers can add things very quickly and accurately, you'd think they'd be good at integration. In this chapter, we'll look at a couple of ways computers can help you deal with tricksy integralses. Techniques like this are called "numerical integration", since they compute the approximate values of definite integrals by using numbers, instead of finding an exact, symbolic, value.

Gottfried Wilhelm von Leibniz, *circa* 1700.

Source: Wikimedia Commons

10.2. Integrals

Imagine that you own a power company[1]. Being a good businessman, you keep a close eye on how much power your customers are using. Over the last year, you've observed that there's a lot of variation in power consumption over the course of a day. Figure 10.1 shows some of the data you've collected.

Kernkraftwerk Beznau, a nuclear power plant in Aargau, Switzerland. As a graduate student, I used to take a shortcut through the plant's parking lot every day on my way to and from work. I wonder how the guards there today would react to a scruffy, backpack-laden student walking by in the night. Would they wave and say "*guten abend!*" as they did to me?

Source: Wikimedia Commons

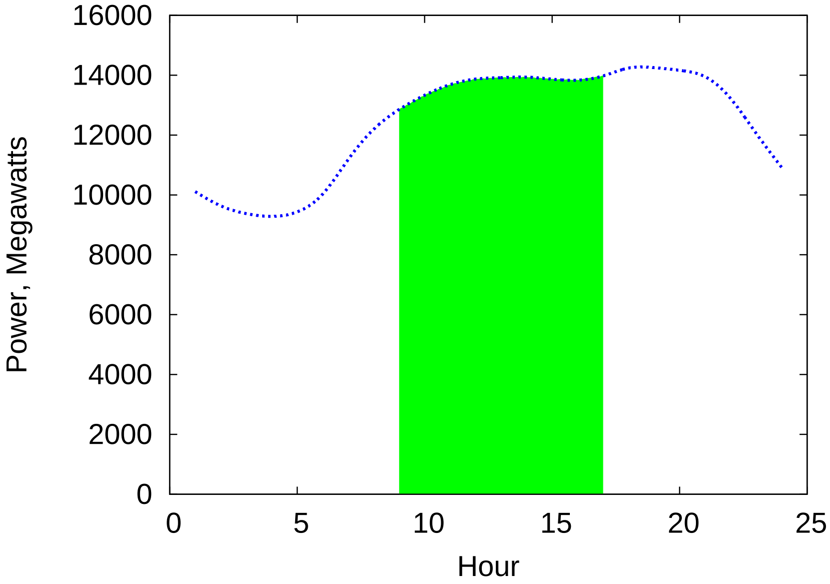


Figure 10.1: Average hourly power consumption on a New England power grid. The shaded region shows times between 9 am and 5 pm.

Source: eia.gov

In the early morning, people are asleep and power usage is low. Usage picks up during the day, falls off a little during commuting time, then surges at night as people turn on their lights, TVs, ovens, and popcorn poppers.

Figure 10.1 shows power usage on the vertical axis and hour of the day on the horizontal axis. Power is the rate at which energy is flowing, and it's measured in watts (or megawatts in this case). This *power* data is interesting, but if we want to know how much coal or gas or sunlight our energy company needs, we have to know how much *energy* people are consuming. How can we determine that? To figure that out, let's start with a simpler example.

A microwave oven might draw 1 kilowatt of power while it's running. If the microwave runs for half an hour, the amount of energy it uses is:

$$1 \text{ kilowatt} \times \frac{1}{2} \text{ hour} = 0.5 \text{ kilowatt-hours}$$

[1] Maybe you do. How would I know?

Kilowatt-hours is a unit of energy. When we do the calculation above, it's equivalent to determining the shaded area shown in Figure 10.2. Mathematically, we could express it like this:

$$E = P(t)\Delta t$$

where E is energy, P(t) = 1000 watts (a steady, unvarying power consumption), and Δt is the amount of time the microwave oven is running.

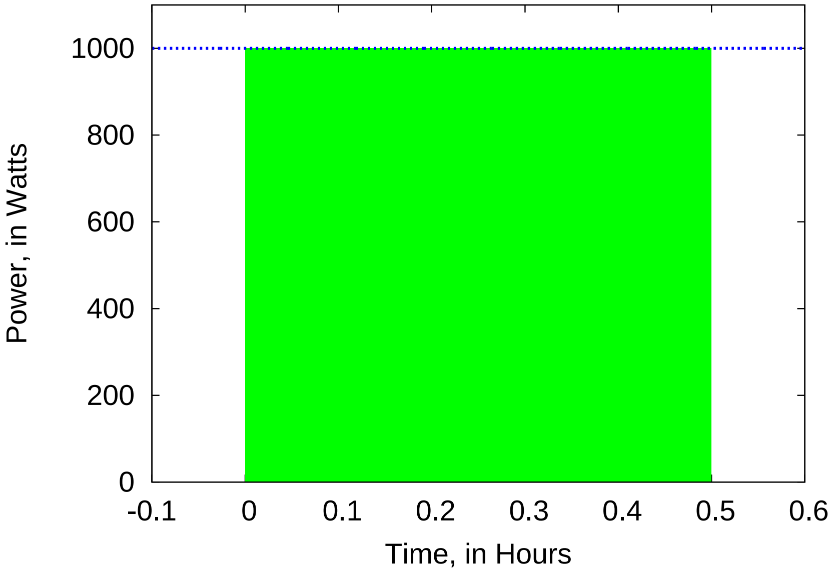


Figure 10.2: If we assume that the microwave oven uses a constant amount of power while it's running, the energy it consumes is equal to the shaded area in this graph, which is the power (P, in watts) multiplied by the amount of time (Δt, in hours).

Unfortunately for our power company, Figure 10.1 shows that our customers don't use the same amount of power all the time, so we can't just do a simple multiplication to find out how much energy they use. If we want to know how much energy is used between 9 am and 5pm, we need to determine the size of the shaded area in Figure 10.1. Mathematically, that's equivalent to evaluating the following integral equation:

$$E = \int_{9am}^{5pm} P(t)dt$$

The important thing to remember is that the integral is just the area under the curve defined by the P(t) function. As we saw in the microwave oven example, this is sometimes trivial to calculate. In calculus class we learn some mathematical tricks to evaluate the integrals of more complicated functions.

Scottish inventor and entrepreneur James Watt, most famous for the invention of the steam engine, for whom the unit of power is named.

Source: Wikimedia Commons

In the following, we'll see how to use a computer program to estimate the value of such integrals without using any tricks. To do so, we'll combine a little modern computing power with the basic principles that Newton and Leibniz used when they invented calculus way back in the 1600s.

The integral here is called a "definite integral" because it only covers a range between two limits (9 am and 5 pm in this case). The techniques we'll talk about in this chapter are all for finding approximate values for definite integrals.

10.3. Slicing up the Problem

If Newton and Leibniz had been familiar with microwave ovens and power plants, they'd have approached the problem this way: slice up the area into manageable bits.

The inventors of calculus realized that the area under a curve could be approximated by the total area of a row of rectangles, as in Figure 10.3, like slices from a loaf of bread. As in the microwave oven example, the area of each rectangle is easy to calculate.

We can slice a function up so that each slice is approximately rectangular.
Source: *Wikimedia Commons*

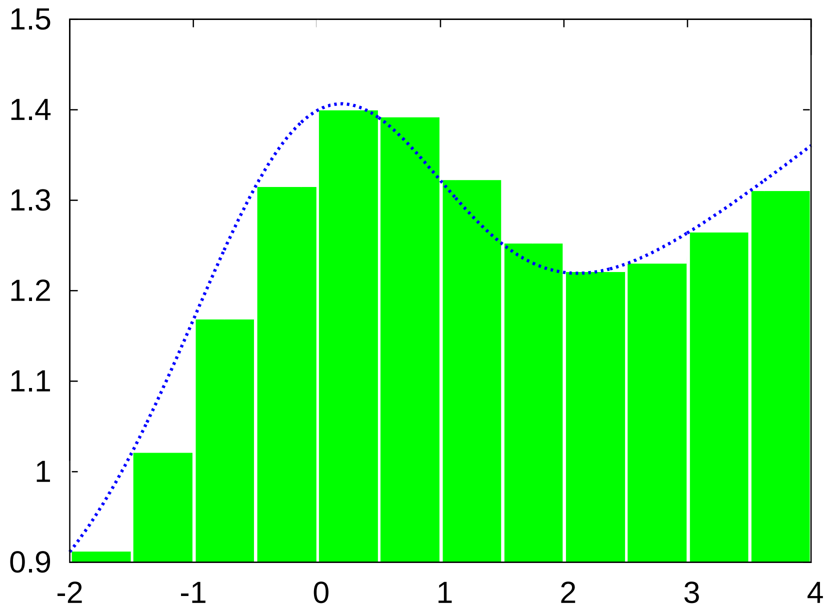


Figure 10.3: The area under the smooth curve is approximately the same as the total area of a row of rectangles of various heights.

If we want a better approximation, we can just use thinner slices, as in Figure 10.4. Mathematicians have found that in some cases we can arrive at the exact area under the curve by mathematically determining what the area *would* be if the width of the slices went to zero. Sometimes we can't calculate this limit, though. In those cases, we have to be satisfied with an approximation, but that's not so bad, because we can often make our approximation as accurate as we want by choosing the width of our slices.

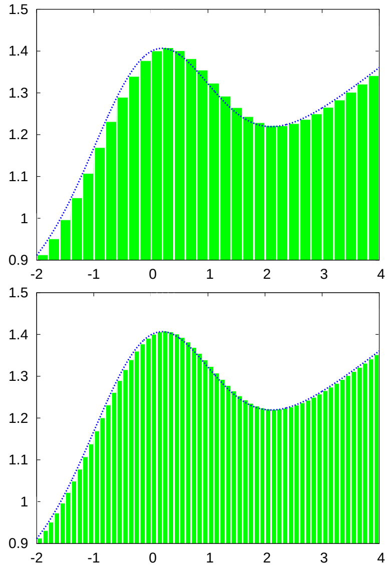


Figure 10.4: More slices give a better approximation.

Now that you have some experience writing programs, you can probably already see how we might do this with a computer. If we wanted to add up the areas of the rectangles shown in Figure 10.5 we could use a "for" loop. It might look something like this:

```
deltax = (xmax-xmin)/nslice;
x = xmin;
for ( i=0; i<nslice; i++ ) {
  height = f(x);
  area += deltax * height;
  x += deltax;
}
```

where nslice is the number of slices, and deltax is the width of each slice. There are several different ways we could make the slices, but for now let's put the upper left-hand corner of each rectangle so that it just touches the curve we're trying to approximate. The function f(x) gives the height of the curve at each value of x. The height of the first rectangle is f(xmin), so its area is f(xmin) times deltax. Each time around the loop we add the area of the current rectangle to to the total area, then move right by a distance deltax and do it again, until we get to the last rectangle.

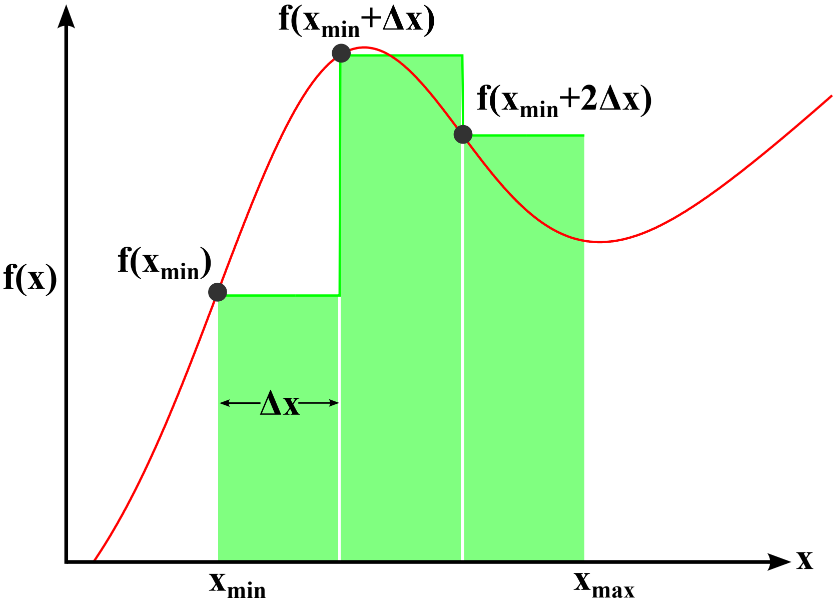


Figure 10.5: Approximating the area under the curve, between xmin and xmax, with three rectangles of width Δx.

Let's apply this technique to our power company. Imagine that we've asked our minions to give us a data file containing two columns of data: The hour of the day (in 24-hour time) and the average amount of power used at that hour (in megawatts). The file might look like Figure 10.6.

Program 10.1 is designed to read this file and estimate the area under the power curve between 9 am and 5 pm. Notice that we've placed the slices as shown in Figure 10.5, which means that each rectangle starts on the hour and covers the time until the next hour begins. This means that we don't want to include the 5:00 measurement (hour 17) in the total area, since we're assuming that this measurement is an estimate of the power used between 5 pm and 6 pm, which is outside the range we're interested in. (See Section 10.5 for more about this.)

Figure 10.6: The contents of the file "`power.dat`", containing hourly power measurements from our power company:

```
 1   10110.66
 2    9636.32
 3    9376.79
 4    9283.72
 5    9433.38
 6   10025.68
 7   11181.30
 8   12193.26
 9   12855.83
10   13332.67
11   13685.37
12   13871.35
13   13918.52
14   13935.75
15   13867.86
16   13833.21
17   13976.59
18   14238.24
19   14253.39
20   14163.90
21   13948.21
22   13220.34
23   12059.76
24   10920.85
```

Also notice that, even though each slice has a width of 1 hour, the program goes ahead and explicitly multiplies by this width, just to make it clear that we're calculating the area of the rectangular slice by multiplying its height times its width. If our data were at half-hour intervals, we'd change the 1.0 to 0.5.

Program 10.1: power.cpp

```
#include <stdio.h>
int main () {
  int hour[24];
  double power[24];
  FILE *input;
  int i;
  double area=0.0;

  input = fopen("power.dat","r");
  for ( i=0; i<24; i++ ) {
    fscanf( input, "%d", &hour[i] );
    fscanf( input, "%lf", &power[i] );
  }
  fclose ( input );

  for ( i=0; i<24; i++ ) {
    // Don't include the 5pm hour:
    if ( hour[i] >= 9 && hour[i] < 17 ) {
      area += power[i]  * 1.0; // 1 hour.
    }
  }
  printf ( "Total energy from 9am-5pm is %lf Mw-hours\n", area );
}
```

Exercise 51: Power to the People!

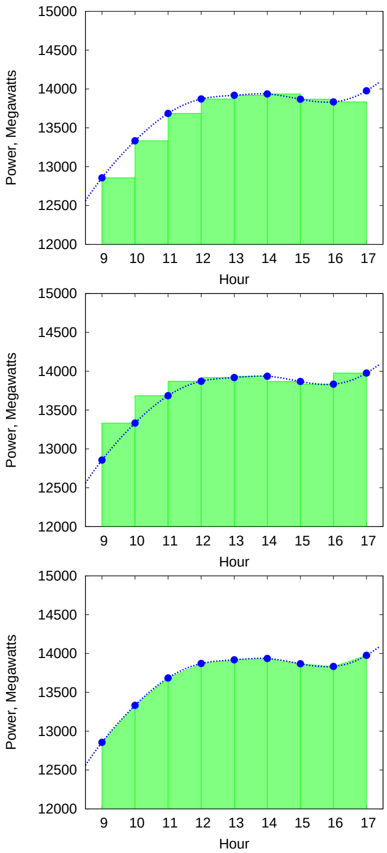
Using *nano*, create the file `power.dat` by entering the data from Figure 10.6. Create, compile and run Program 10.1. Make a note of the result, for later use.

By looking at Figure 10.1, make a rough estimate of the 9 to 5 power usage by multiplying the curve's approximate height, in megawatts, by 8 hours (the interval between 9 am and 5 pm). Is your rough estimate consistent with Program 10.1's estimate?

10.4. Trapezoids

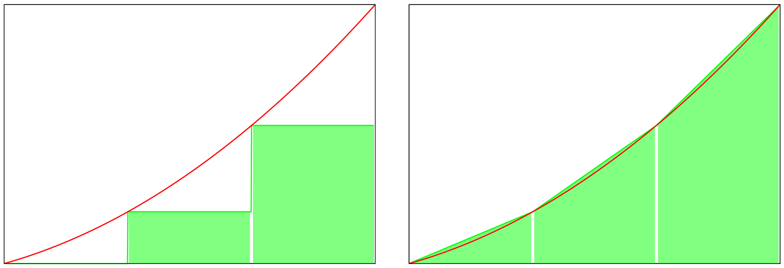
The technique above would probably work fine if we used enough slices, but as you can see from Figure 10.5 the rectangles might not fit the curve very well if we only use a small number of slices, or if the curve goes up and down a lot over distances narrower than the width of a slice.

Figure 10.7:
Top: Right Riemann Sum
Middle: Left Riemann Sum
Bottom: Trapezoid Sum



In the preceding section, we placed our rectangles on the right-hand side of the curve. Mathematicians call a this a "Right Riemann Sum". We could alternatively have chosen to put them on the left-hand side, as shown in the middle graph of Figure 10.7 (called a "Left Riemann Sum"). Neither of these rectangular sums fits the curve particularly well, but what if we averaged the two? That's what's shown in the bottom graph.

The slices in the last graph aren't rectangles. Instead, they're "trapezoids". A trapezoid is a four-sided figure with two or more parallel sides (unlike a rectangle which must have two pairs of parallel sides). We can use trapezoidal slices to estimate the area under the curve much more efficiently than with rectangular slices. These new slices are the same width as the rectangular ones, but their top edge has *both* corners, right and left, on the curve.



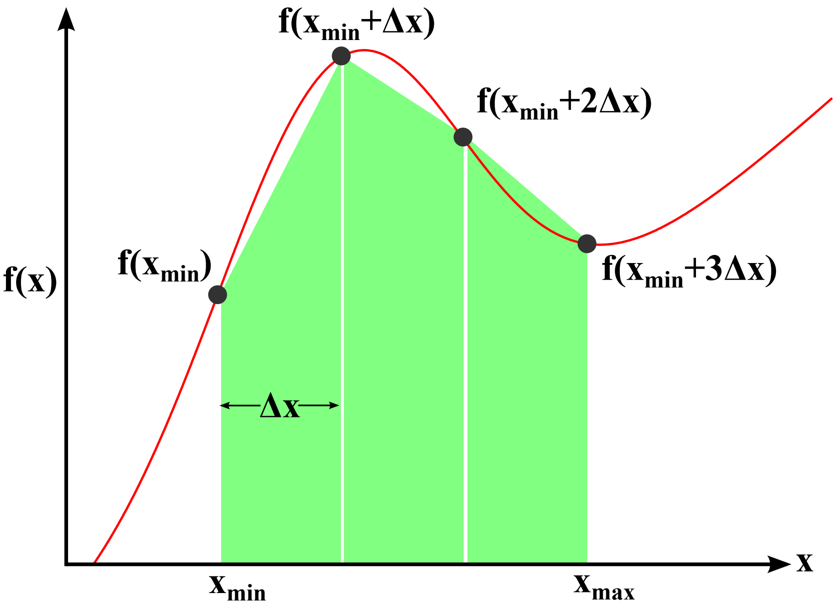


Figure 10.8: Approximating the area under the curve, between `xmin` and `xmax`, with three *trapezoids* of width Δx. Compare this with Figure 10.5, which uses rectangles.

Figure 10.8 shows how we might slice up an area into trapezoidal sections. As before, we can add up the areas of the slices to get an estimate of the area under the curve, but to do this we'll first need to know how to find the area of a trapezoid.

It turns out that this isn't so hard. The area of a rectangle is its height times its width, but the area of a trapezoid is its *average* height times its width. You can see why this is so by looking at Figure 10.9.

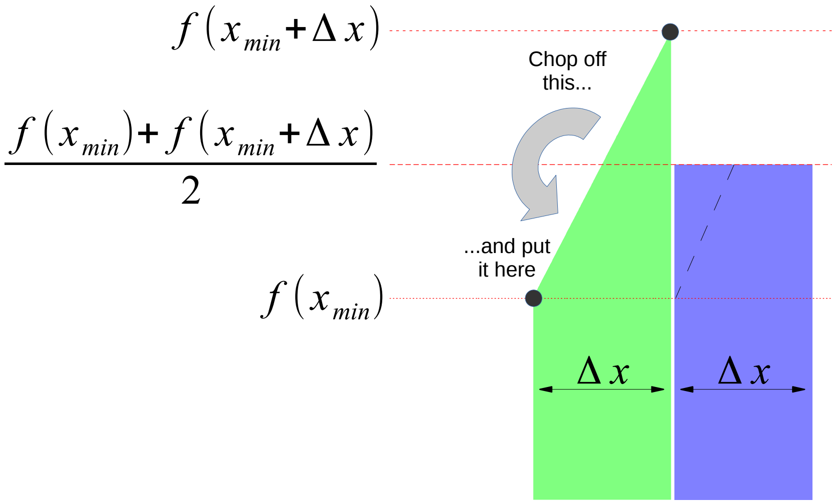


Figure 10.9: How to find the area of a trapezoidal slice.

Once we know how to find the area of a trapezoid, it's easy to modify Program 10.1 so that it uses this shape instead of rectangles. We only need to change one line. The result is Program 10.2

Program 10.2: power.cpp, with trapezoids instead of rectangles

```
#include <stdio.h>
int main () {
  int hour[24];
  double power[24];
  FILE *input;
  int i;
  double area=0.0;

  input = fopen("power.dat","r");
  for ( i=0; i<24; i++ ) {
    fscanf( input, "%d", &hour[i] );
    fscanf( input, "%lf", &power[i] );
  }
  fclose ( input );

  for ( i=0; i<24; i++ ) {
    // Don't include the 5pm hour:
    if ( hour[i] >= 9 && hour[i] < 17 ) {
      area += 0.5 * (power[i] + power[i+1]) * 1.0; // 1 hour.
    }
  }

  printf ( "Total energy from 9am-5pm is %lf Mw-hours\n", area );
}
```

The hammer dulcimer has a trapezoidal shape. The American folk band "Trapezoid" took its name from the shape of this instrument.

Source: *Wikimedia Commons*

Program 10.2 multiplies the width of each trapezoid by its average height. The average height is:

$$\frac{\texttt{power[i]} + \texttt{power[i+1]}}{2}$$

This way of approximating the value of an integral is called the "trapezoid rule".

Exercise 52: More Power to You!

Modify your `power.cpp` program so that it looks like Program 10.2. Compile and run it. How does the result compare with the result from the previous version? Does this agree with what you'd expect after looking at Figure 10.7?

10.5. Fencepost Problems

Consider the cheery scene below. It shows a section of fence with six panels. Notice that this requires *seven* fenceposts.

When we look at data we need to think about whether we're interested in the measurements themselves or the intervals between them. There will always be one more fencepost than the number of panels, so if we're interested in the intervals between measurements we need to be careful not to overcount.

Imagine that each of the fenceposts represents a measurement of our power plant's output at a particular time of day. Notice that Programs 10.1 and 10.2 are careful to stop *before* the 5pm measurement. These programs add up the energy produced during the intervals between measurements.

This subtlety occurs often in programming. It's called a "fencepost problem". Whenever you write a program that works its way through a number of measurements, always stop and think about whether you're interested in the measurements or the "in-betweens".

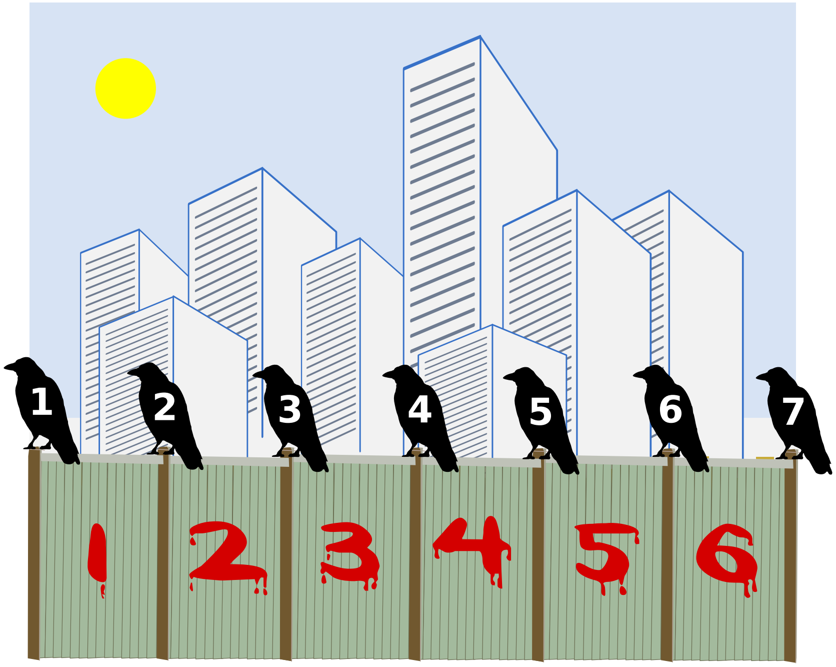


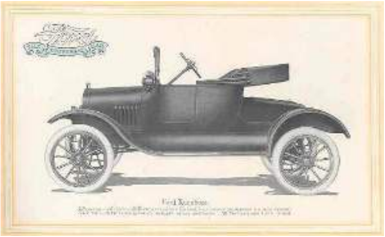
Figure 10.10: A fencepost problem. Are we interested in the number of panels or the number of crows?

10.6. Uneven Slices

In the preceding examples we've assumed that our measurements were evenly spaced. What if they're not, though? To explore that possibility, let's take a road trip!

In October 1925 Harry Lillis "Bing" Crosby and his pal Al Rinker set out from Spokane, Washington, bound for Los Angeles in Al's beat-up Ford Model T. Over the next three weeks they made their way down the coast, stopping whenever the car broke down and working a day or two to earn some money. The car survived until the outskirts of LA, where its engine finally died. They were taken in by Al's sister, the singer Mildred Bailey, who helped the boys find jobs performing in LA. Within a year they were hired by internationally-known bandleader Paul Whiteman. The rest, as they say, is history.

Source: Wikimedia Commons



Your car's odometer tells you how far you've driven, but modern cars don't measure distance directly. Instead, they use your velocity and a little bit of calculus to determine how many miles you've gone.

A mathematician would say that velocity is the time derivative of position, and she might write that relationship like this:

$$v = \frac{dx}{dt}$$

where dx represents a small change in position, dt is a small change in time, and v is the velocity. If the velocity doesn't change, then this is the same as saying that velocity is equal to distance divided by time. If we know the velocity and the time, we can calculate the distance as $distance = velocity \times time$.

If the velocity isn't constant, things get more complicated. In that case, our mathematician friend would tell us that we could find the distance like this:

$$\text{distance} = \int_{t_0}^{t_1} v(t)dt$$

where $v(t)$ is a function that tells us the velocity at a given time. The times t_0 and t_1 are when our trip starts and ends, repectively.

That looks complicated, but it's just like what we've been doing earlier in this chapter. If we're given some data about the car's velocity at various times, we can write a computer program to do the integral above and estimate the distance we've traveled.

Imagine we're going on a long trip. When we start, our velocity is zero. Then we pull out onto the highway and start driving. We can't drive at a constant speed, though. Sometimes traffic will slow us down. Sometimes we'll be driving on roads with higher or lower speed limits. Sometimes we'll be zoned out listening to our favorite tunes and find that we've spent the last ten miles drifting along behind a sputtering jalopy. Eventually, we'll reach our destination and our velocity will go back to zero again.

If we noted the time and our speed occasionally, the data might look like the large dots in Figure 10.11. We're lazy and easily distracted, so we haven't done the measurements at regular intervals.

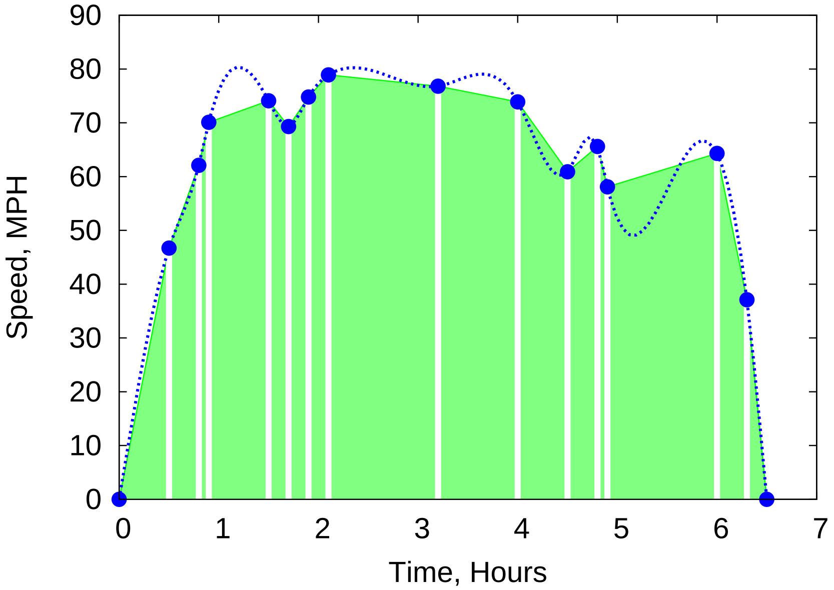


Figure 10.11: The large dots represent measurements of our speed. If we could have constantly monitored the speed it might have looked like the dashed curve. The whole trip takes about six and a half hours.

As you can see from the figure, we can draw trapezoids that connect the dots, just as we did with the power plant data. The only difference is that these trapezoids aren't all the same width. That's not a problem. We just need to take the width into account when we calculate the area of each trapezoid. Adding all of the areas together gives us the total area of the shaded region in Figure 10.11. The size of this shaded area is approximately equal to the distance we've traveled, as given by the integral equation our mathematician friend gave us above.

```
0.0 0.0
0.5 46.7
0.8 62.1
0.9 70.1
1.5 74.1
1.7 69.3
1.9 74.8
2.1 78.9
3.2 76.8
4.0 73.9
4.5 60.9
4.8 65.6
4.9 58.1
6.0 64.3
6.3 37.1
6.5 0.0
```

Figure 10.12: The file `roadtrip.dat`, used by Program 10.3. The first column is time, in hours. The second column is our speed, in miles per hour, at that time.

Program 10.3 does the work for us. It's similar to Program 10.2, but it doesn't assume that there's a particular number of measurements. The power plant program knew that there were 24 hours in a day, so it could assume there would be 24 measurements to read from its data file. On our road trip we just took some measurements at random times. On the next trip we might take more or fewer. Because of this, the new program doesn't start by reading all of the data into a fixed-size array. Instead, it uses a different strategy that just focuses on measurements one or two at a time, as they're read from the data file.

For each of the trapezoids in Figure 10.11 we need two pairs of time and speed measurements[2]. The difference in the two times tells us the width of the trapezoid. The average of the two speeds tells us the average height of the trapezoid. Program 10.3 waits until it has read the first two lines from the data file, then does these calculations to find the area of the first trapezoid. Then it continues on, doing the same for each subsequent pair of data points. Notice that, at any point in our

[2] See Section 10.5.

trip, the area we've accumulated so far will be equal to the distance we've traveled so far. To emphasize this, the program prints out the updated time and distance each time it reads a new data point from the file.

Program 10.3: roadtrip.cpp

```
#include <stdio.h>
int main () {
  double hour, velocity;
  double oldhour, oldvelocity;
  double height, width;
  double area=0.0;
  int nmeasurements=0;
  FILE *input;

  input = fopen("roadtrip.dat","r");
  while ( fscanf( input, "%lf %lf", &hour, &velocity ) == 2 ) {
    if ( nmeasurements != 0 ) {
      height = 0.5 * (oldvelocity+velocity);
      width = hour - oldhour;
      area +=  height*width;
      printf ( "Distance after %lf hours is %lf miles\n",
               hour, area );
    }
    oldhour = hour;
    oldvelocity = velocity;
    nmeasurements++;
  }
  fclose ( input );

  printf ( "Total distance is %lf miles\n", area );
}
```

Notice that the program uses two variables, `oldhour` and `oldvelocity`, to remember the previous data point's values. To make the program wait until there are at least two data points, we use the variable `nmeasurements` to count the number of points we've read so far, and test this value before we start doing any calculations. After the first trip around the loop, `oldhour` and `oldvelocity` have been set and we're ready to calculate the area of the first trapezoid during the next trip around the loop.

Remember that our result is only an estimate of the distance we've traveled. The estimate would be more accurate if we recorded more speed measurements during our trip. (If we'd only written down the speed at the beginning and end of the trip, the program would tell us that the distance was zero!)

10.7. Integrating Functions

In the exercises above, we've been finding the integral of a curve defined by data points, instead of being defined by some mathematical function. This is one common reason people use numerical integration: they have some data points, but don't know the underlying mathematics that generated them. You can't use calculus to compute the integral of a function if you don't know what that function is!

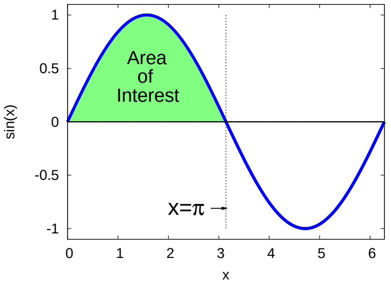


Figure 10.13: sin(x)

But what if you do know the function? As noted before, integration can be hard, and much of what we learn in calculus class is a set of tricks for finding the values of certain integrals. Sometimes, though, there are no applicable tricks that will let write down a value for a particular integral in terms of elementary functions (trigonometric functions, logarithms, *etc.*), or even "special" functions (the error function, the gamma function, *etc.*). Integrals of some seemingly simple expressions like $sin(sin(x))$ turn out to be impossible to evaluate.

Happily, in these cases the poor beleaguered mathematician can turn to numerical integration. As long as we can find the value of a function at any point within the range we're interested in, we can numerically approximate the value of the definite integral of the function over that range.

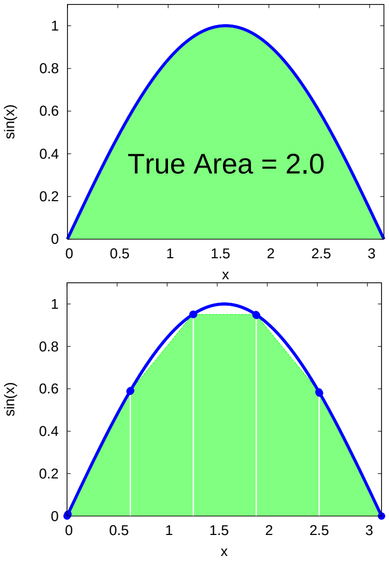


Figure 10.14: The integral of the sine function between 0 and π has a value of 2 (the area under the curve in the top graph). The bottom graph shows this area approximated by five trapezoidal slices.

Let's look at an example where we *can* find the value of the integral mathematically. That will allow us to compare an exact mathematical solution to an approximate solution computed by the trapezoid rule.

For example, in calculus class we learned how to integrate the sine function (see Figure 10.13):

$$\int_A^B sin(x)dx = cos(A) - cos(B)$$

This tells us that the area under the sine curve between 0 and π is exactly $cos(0) - cos(\pi) = 2$, as shown in the top graph of Figure 10.14. Let's use the trapezoid rule to find this same area, and see how close it gets to the true answer.

That's what Program 10.4 does. Notice that, instead of using `sin(x)` directly, the program uses a function we define ourselves, named `func`. If we ever want to use this program to integrate something other than `sin(x)`, we'll only need to change this function definition. We can put anything we want to inside `func`. It could be as simple as `sin(x)` or as complicated as the "Red Baron" flight path we used in Chapter 9.

Program 10.4: integrate.cpp

```
#include <stdio.h>
#include <math.h>

double func( double x ) {
  double value;
  value = sin(x);
  return(value);
}

int main () {
  double x, delta, area=0;
  double height;
  double xmin=0.0, xmax=M_PI;
  int i, nsteps=5;

  delta = (xmax-xmin)/nsteps;
  x = xmin;
  for ( i=0; i<nsteps; i++ ) {
    height = ( func(x) + func(x+delta) ) / 2.0;
    area += delta * height;
    x += delta;
  }

  printf ( "Integral from %lf to %lf is %lf\n",
           xmin, xmax, area );
}
```

Slicing an onion in preparation for integrating it into dinner.

Source: Wikimedia Commons

The program specifies how many slices we want to use by setting the value of `nsteps`. (It's set to 5 here.) The variables `xmin` and `xmax` set the range over which we want to integrate. Notice that we use the symbol `M_PI` from `math.h` to set `xmax`, so we don't have to type out a long string of π's digits.

Most of the program's work is done in the "`for`" loop that works its way through the slices, one at a time, adding each area to the total area. As we go through the slices, we need to keep track of where we are on the x axis. Before we start the "`for`" loop, we set the value of `x` to `xmin`, and then we add the slice width (`delta`) to this when we're ready to go to the next slice.

To find the area of each slice, we multiply its average height by its width (`delta`). See Figures 10.8 and 10.9.

Exercise 53: Sines of the Times

1. Create, compile, and run Program 10.4. How close does its approximation come to the true value of the area?

2. Now change `nsteps` to 10, recompile, and run again. Is the answer closer to the true value? Try `nsteps` values of 100 and 1000. Does the program take noticeably longer if you use these large values for `nsteps`?

3. Now modify your program so that, instead of `sin(x)`, it finds the area under the curve `sin(sin(x))` between 0 and π. (See Figure 10.15.) Remember that it's not possible to find an exact value for this integral mathematically. Does the computer have any trouble finding an approximate solution? Write down this solution for later.

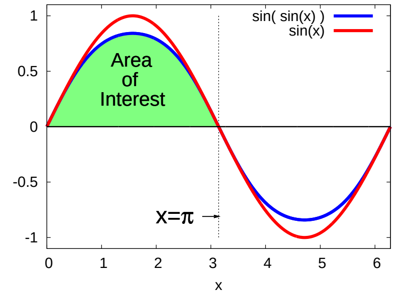


Figure 10.15: $sin(sin(x))$ and $sin(x)$

So far, this looks pretty good! We can come up with approximate values even for integrals that are impossible to solve exactly. Will this always work? Unfortunately, there are some pitfalls to look out for.

Figure 10.16 illustrates one potential problem. If the function we're integrating varies rapidly compared to the size of our slices, we may miss important features, possibly causing our estimate of the area to differ greatly from the true value.

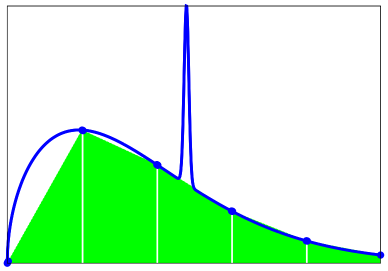


Figure 10.16: When integrating a function with sudden spikes or dips, we need to be careful to use a small enough slice width so that we avoid "stepping over" interesting features and missing them.

Imagine what would happen if the spike in Figure 10.16 was very high. Even worse, what if the function goes to infinity at some values of x? Even mundane functions like $tan(x)$ or $1/x$ have infinities that we need to loook out for. The tangent of x goes to positive infinity as we approach $\pi/2$ from the right, and to negative infinity as we approach from the left!

If our program tried to find the value of the function at one of these x values, it would crash, but if we avoid these x values we may be missing a large part of the area we're trying to estimate. The area in the region around such points might even be infinite!

The trapezoid rule works well for a wide range of well-behaved functions, but it's important to be aware of the shape of the function you're integrating, and look out for problems like this. Even with ill-behaved functions, though, we can still use the trapezoid rule to find the integral over regions that don't include problematic points. We could, for example, integrate $tan(x)$ between 0 and 1 with no problem.

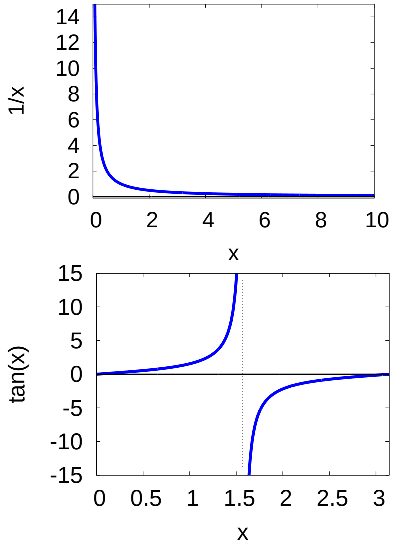


Figure 10.17: Functions like $1/x$ and $tan(x)$ go to infinity at some values of x. Our program would obviously have trouble if we included these x values in the range over which we tried to integrate.

But what about...?

Wait a minute. Doesn't Program 10.4 calculate `func(x)` for the same values of `x` multiple times?

Take a look at Figure 10.8 again. If Program 10.4 were calculating the area of the first slice it would use `func(xmin)` for the height of the trapezoid's left-hand side and `func(xmin+delta)` for the right-hand side. For the second slice, it would use `func(xmin+delta)` for the left and `func(xmin+2*delta)` for the right. On the third slice the program would use `func(xmin+2*delta)` for the left and `func(xmin+3*delta)` for the right.

It takes the computer some time to process the statements inside a function. The program would run faster if we didn't have to calculate each of the middle `func` values twice. If `func` were more complicated, or if we had a lot of slices, the amount of time saved could be large.

We can do a little algebra and eliminate the duplication. If we have n slices we could write the sum of their areas like this:

$$\begin{aligned} A = \Delta x & \frac{f(x_{min}) + f(x_{min} + \Delta x)}{2} + \\ \Delta x & \frac{f(x_{min} + \Delta x) + f(x_{min} + 2\Delta x)}{2} + \ldots + \\ \Delta x & \frac{f(x_{min} + (n-1)\Delta x) + f(x_{max})}{2} \end{aligned}$$

Collecting terms, we can rewrite the equation like this, so that we only calculate the value of $f(x)$ once for each value of x:

$$A = \Delta x \left[\frac{f(x_{min}) + f(x_{max})}{2} + \sum_{i=1}^{n-1} f(x_{min} + i\Delta x) \right]$$

To take advantage of this, we could replace the loop in Program 10.4 with a loop like this:

```
 double sum = 0;
...
  for ( i=1; i<nsteps; i++ ) {
    sum += func(x+delta);
    x += delta;
  }
  area = delta*( (func(xmin) + func(xmax))/2.0 + sum );
```

There! We fixed it.

10.8. Negative Areas

Summer is here and it's time to fill the ol' backyard pool! Let's turn on the faucet. How long will it be before we can dive in? If we know how fast water is going into the pool, we should be able to calculate how much water has accumulated after a given amount of time. If the flow rate is given by a function $r(t)$, then the amount of water at time t is:

$$\text{amount of water} = \int_{t_0}^{t} r(t)dt$$

The function $r(t)$ might look like the blocky solid line in the top graph of Figure 10.18.

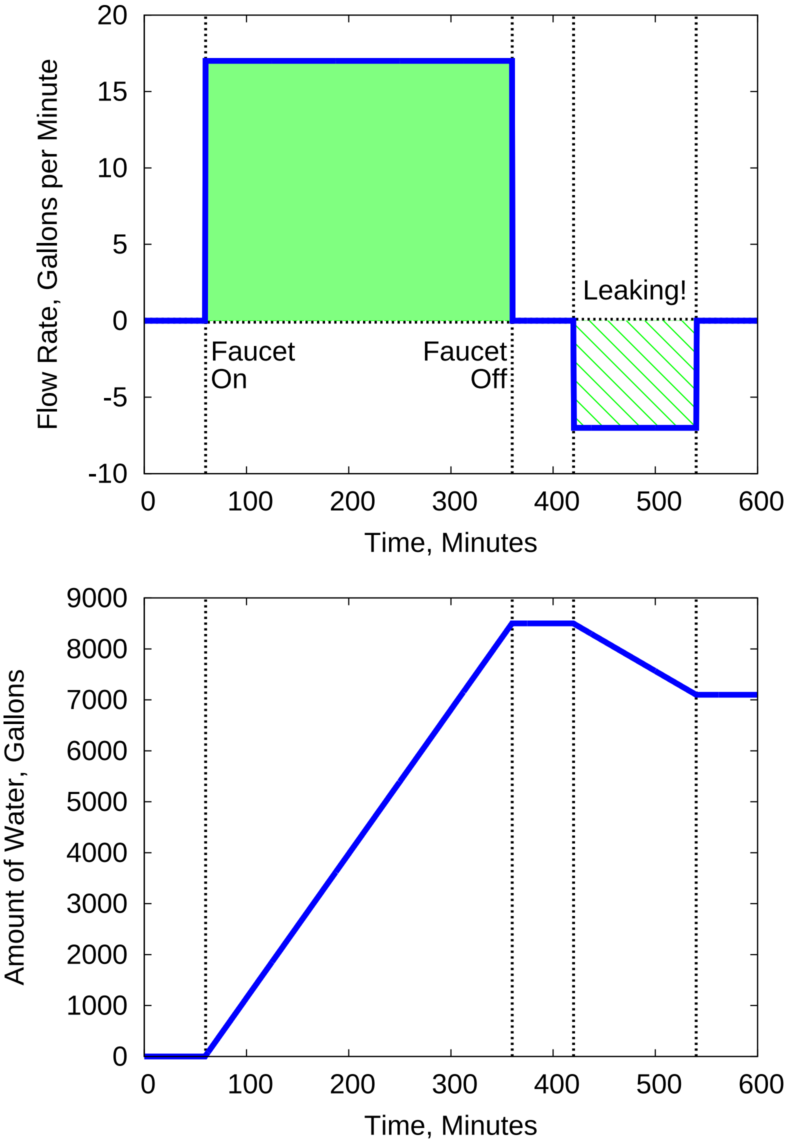


Figure 10.18: Filling a pool. The top graph shows the flow rate. Notice that a leak corresponds to a negative flow rate (water is flowing out of the pool instead of into it). The bottom graph shows how much water has accumulated in the pool Right before the leak the pool contained about 8,000 gallons of water, but the leak drained some of that off.

The amount of water in the pool is given by the integral of this curve, which is just the shaded area between the curve and the x axis in the top graph of Figure 10.18.

This figure shows the pool being filled for a while at a constant rate of 17 gallons per minute. Eventually, we accumulate about 8,000 gallons of water, as shown in the bottom graph, which shows the integral of $r(t)$ up to a given time.

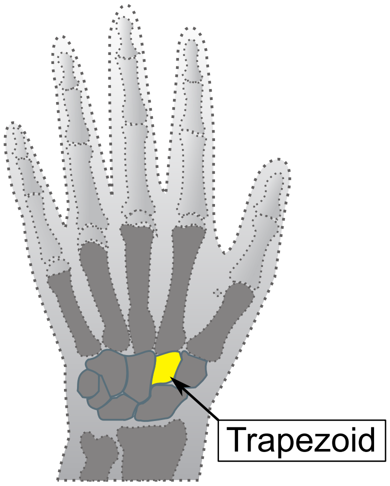
Uh oh! after filling the pool, it sprang a leak! This let water run out of the pool at a rate of 7 gallons per minute until we found the leak and patched it. If $r(t)$ is the rate of water *going into* the pool, that means that $r(t)$ had a *negative* value while the pool was leaking, as shown in the top graph of Figure 10.18.

If we used a computer program to slice up this curve and calculate its area, we'd find that the leak would contribute a negative amount to the sum because the height of the slices in this region would be negative. This is actually what we expect: The first part of the curve shows water flowing into the pool, and the second part of the curve shows water flowing out. To find out how much water we have at the end, we need to subtract the water that escaped through the leak.

The important thing to know is that we don't have to modify our programs in any way because of this. It just gets taken care of automatically when we multiply the height of a slice times its width. When the function we're integrating has a negative value, it contributes a negative amount to the total area.

10.9. A General-Purpose Trapezoid Integration Function

We can use the trick we learned in Section 9.17 of Chapter 9 to create a general-purpose function that can integrate anything using the trapezoid rule. Take a look at the function `trapint` defined below:



You carry a "trapezoid bone" in your wrist.

Source: Wikimedia Commons

```
double trapint ( double (*f)(double),
                 double xmin, double xmax,
                 int nsteps ) {
  double x, delta, area=0;
  double height;
  int i;

  delta = (xmax-xmin)/nsteps;
  x = xmin;
  for ( i=0; i<nsteps; i++ ) {
    height = ( f(x) + f(x+delta) ) / 2.0;
    area += delta * height;
    x += delta;
  }

  return( area );
}
```

"`trapint`" estimates the integral of the function "`f`" between `xmin` and `xmax` using the trapezoid rule. The argument `nsteps` specifies the number of trapezoids.

Notice that the first argument given to `trapint` is the name of the function we want to integrate. We could say `cos`, for example, to compute the area under some section of the cosine function, or `sqrt` to do the same for the square root function. But we aren't limited to the built-in functions. We also use `trapint` with a function we write ourselves. The only restriction is that the function we use must return a `double` value and take one `double` argument.

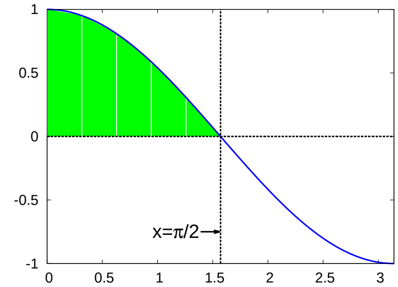


Figure 10.19: The integral of $cos(x)$ between zero and $\pi/2$, approximated by trapezoids.

Here's how `trapint` might be used in a program:

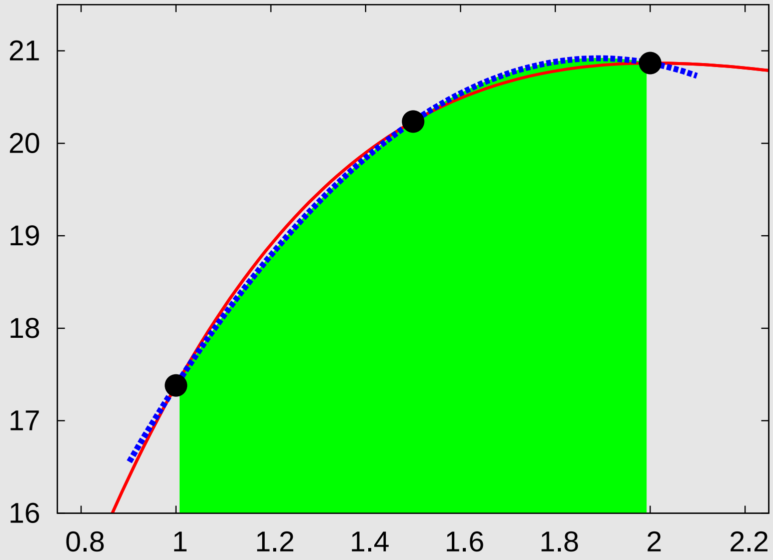
```
area = trapint( cos, 0, M_PI/2.0, 5 );
```

The line above would calculate an estimate of the area under the cosine function between zero and $\frac{\pi}{2}$ radians, using five trapezoidal slices. (See Figure 10.19.)

But what about. . . ?

Are trapezoids the best possible shape for numerical integration? Not necessarily. There are other techniques that work better in some circumstances. Sometimes these techniques will give a more accurate estimate of the area. Sometimes they'll give you an estimate more quickly.

One common alternative to the "trapezoid rule" is called "Simpson's rule", named after 18*th*-Century British mathematician Thomas Simpson. Instead of connecting two data points with a straight line to make the top of each slice, Simpson's rule draws a section of a parabola through *three* adjacent points.



In the graph above, the dashed line represents a section of a parabola that approximates the shape of the curve. This parabola forms the top of slice. As you can see, Simpson's rule often fits a curve better than the trapezoid rule. The area of the slice can be determined from the parameters of this parabola and the width of the slice.

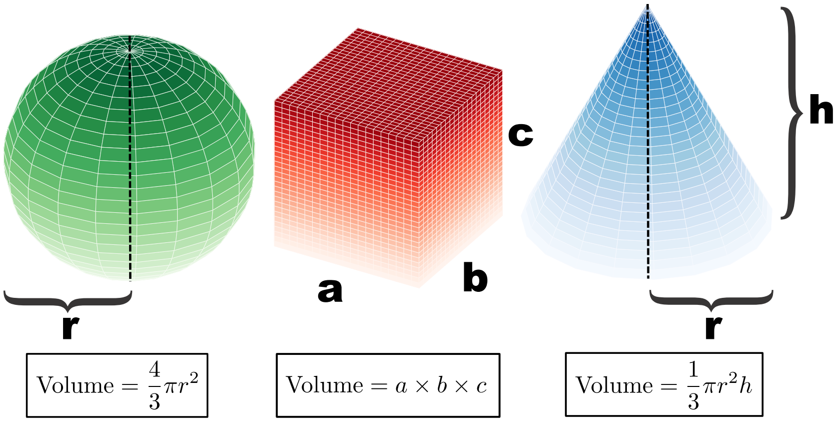
There are many other numerical integration schemes. One of them is the Monte Carlo method we'll talk about in a later section of this chapter. It doesn't use slices at all!

Thomas Simpson, 1710-1761.

Source: University of St. Andrews

10.10. Estimating Volume

The integrals we've used so far measure 2-dimensional areas. Sometimes we'll also need to estimate 3-dimensional volumes. We know nice mathematical formulas that tell us the volume of some simple geometrical shapes: a sphere, a rectangular box, or a cone, for example.


r
a
b
c
h
r
$\text{Volume} = \frac{4}{3}\pi r^2$
$\text{Volume} = a \times b \times c$
$\text{Volume} = \frac{1}{3}\pi r^2 h$

But sometimes we want to find the volume of a more complicated shape. That's OK. We can estimate the shape's volume even if we don't have a formula that will give us an exact value. One way to do this is by breaking the complicated shape up into simpler shapes, then adding up their volumes. The best way of doing this will depend on the particular shape we're dealing with, but let's look at a technique that will work with one group of common shapes.

Imagine that we're given a shape that's cylindrically symmetric about some axis, like the lovely vase in Figure 10.20. By cylindrically symmetric, I mean that the vase would look the same if we rotated it around on the table. A cylinder or a cone would also be cylindrically symmetric shapes.

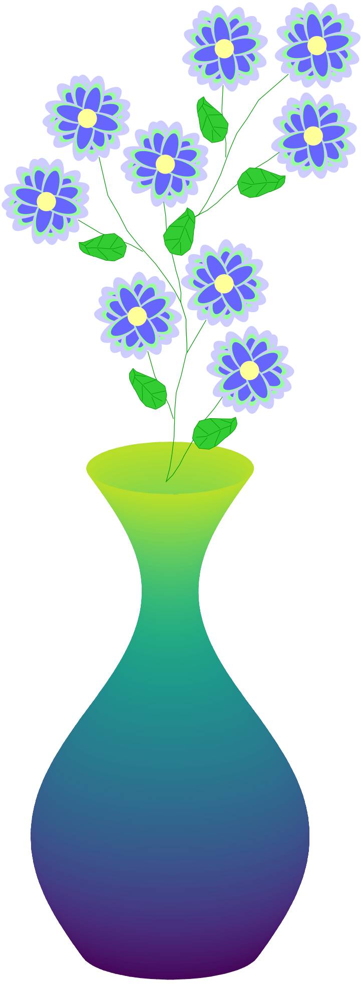


Figure 10.20: This vase has a cylindrically symmetric shape.

Because of the vase's symmetry, we could describe its shape completely by just specifying the shape of its sides. This might be easier to see if we lay the vase on its side, as in Figure 10.21. Now we can see that this is similar to the 2-dimensional problems we did earlier. The curve of the vase's side could be approximately sliced into something like trapezoids, and we could add up the volumes of these shapes to get an estimate of the vase's total volume.

We're not calculating areas here, though. Now we're calculating volumes. The slices of our vase won't be trapezoids, they'll be some 3-dimensional shape. As you can see from Figure 10.22, the slices will be truncated cones.

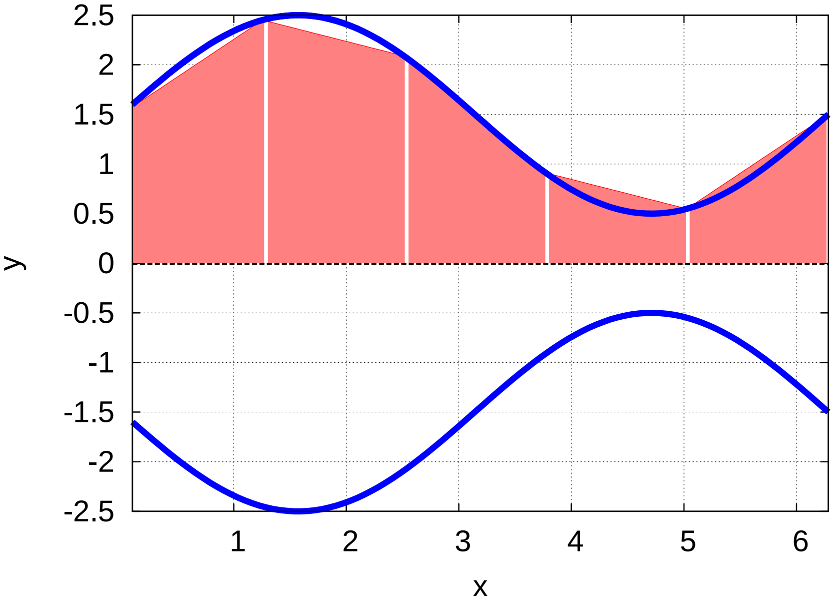


Figure 10.21: We can graph the curves that define the shape of the vase, and approximate them with simpler shapes.

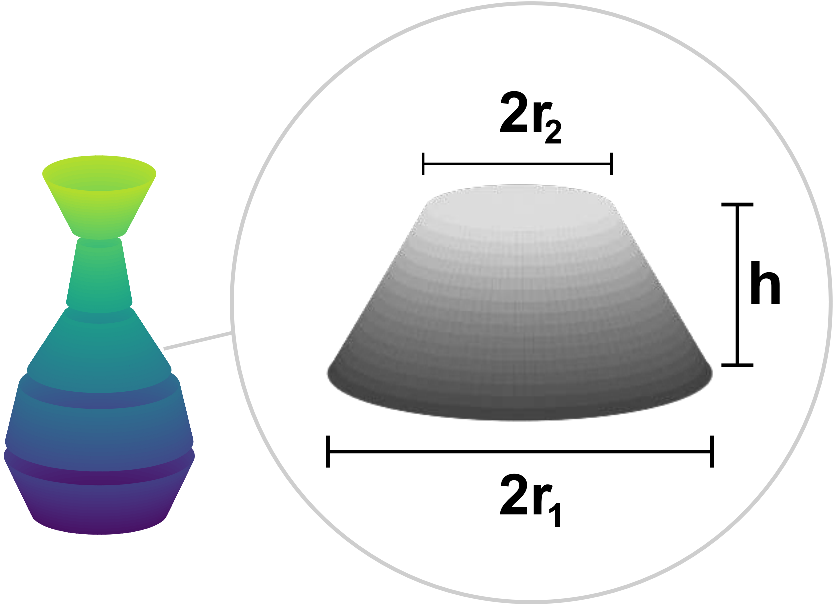


Figure 10.22: The vase's volume could be sliced into a bunch of truncated cones that approximate its shape. The diameter of the top is $2r_2$ and the diameter of the bottom is $2r_1$, where r_2 and r_1 are the radius of top and bottom.

Fortunately there's a formula that will tell us the volume of a truncated cone, given its height, h, and the radius of its top and bottom circles, r_1 and r_2. (Notice that in this case it doesn't matter which is top and which is bottom. The volume would be the same if we flipped the shape over.) The formula is:

$$\text{Volume} = \frac{1}{3}\pi(r_1^2 + r_1 r_2 + r_2^2)h$$

The values of r_1 and r_2 for each truncated cone are given by the height of the curve in Figure 10.21. For a real vase, we could get these values by just measuring the diameter of the vase at a few points. The data might look like this:

```
0.0 3.2
1.2 4.9
2.5 4.1
3.7 1.8
4.9 1.1
6.2 3.0
```

where the first column is the height above the tabletop and the second column is the vase's diameter at that height. If we put these data into a file named `vase.dat` we could write a program like Program 10.5 (`vase.cpp`) to read the file and estimate the vase's volume.

Notice that this program looks a look like our earlier `roadtrip.cpp` program (Program 10.3). It uses the same strategy for reading an unknown number of data points from a file and using the intervals between them. The main difference is that now we're calculating the volumes of truncated cones, using the formula above, whereas in the earlier program we were calculating the areas of trapezoids.

We could use this this new program to estimate the volume of any cylindrically-symmetric shape, based on some measurements of height and diameter stored in a data file.

Program 10.5: vase.cpp, Estimating volume based on measurements

```
#include <stdio.h>
#include <math.h>
int main () {
  double height, diameter;
  double oldheight, olddiameter;
  double h,r1,r2;
  double volume=0.0;
  int nmeasurements=0;
  FILE *input;

  input = fopen("vase.dat","r");
  while ( fscanf( input, "%lf %lf", &height, &diameter ) == 2 ) {
    if ( nmeasurements != 0 ) {
      r1 = olddiameter/2.0;
      r2 = diameter/2.0;
      h = height-oldheight;
      volume += M_PI*( r1*r1 + r1*r2 + r2*r2 )*h/3.0;
    }
    oldheight = height;
    olddiameter = diameter;
    nmeasurements++;
  }
  fclose ( input );

  printf ( "Total volume is %lf\n", volume );
}
```

Sometimes the volume we need to estimate won't belong to a physical object like a vase that we can take measurements from. We might be given a mathematical function that describes a shape in 3-dimensional space, and asked to find the volume inside it. This would be analogous to the 2-dimensional integration of a function that we did earlier, in Program 10.4.

Program 10.6 (`vase-func.cpp`) uses a function to describe the shape of the vase's side, then divides the vase's height up into five truncated-cone-shaped slices and adds up their volumes. It works just like Program 10.4, but calculates the volumes of truncated-cone-shaped slices instead of the areas of trapezoids. The function `func` at the top defines the shape of the vase's side, and the values `xmin` and `xmax` are the bottom and top of the vase, respectively.

Again, this program could be used to estimate the volume of any cylindrically-symmetric shape. We'd just need to change the definition of `func` and `xmin` and `xmax` appropriately.

Program 10.6: vase-func.cpp, Estimating volume based on a function

```
#include <stdio.h>
#include <math.h>
double func( double x ) {
  double value;
  value = 1.5 + sin(x);
  return(value);
}
int main () {
  double x, volume=0;
  double xmin=0.1, xmax=2.0*M_PI;
  double r1, r2, h;
  int i, nsteps=5;

  h = (xmax-xmin)/nsteps;
  x = xmin;
  for ( i=0; i<nsteps; i++ ) {
    r1 = func(x);
    r2 = func(x+h);
    volume += M_PI*( r1*r1 + r1*r2 + r2*r2 )*h/3.0;
    x += h;
  }

  printf ( "Integral from %lf to %lf is %lf\n",
           xmin, xmax, volume );
}
```

10.11. Monte Carlo Integration

Figure 10.23: Chickens, finding the approximate value of an integral.
Source: Wikimedia Commons

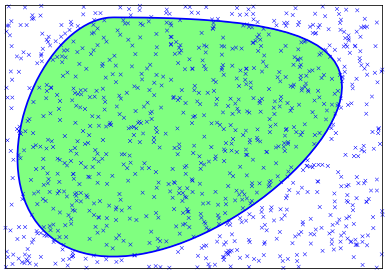


Figure 10.24: The ratio of pecks inside the shape to pecks outside the shape is approximately equal to the ratio of the shape's area to the area of its enclosure.

Did you know that chickens can do calculus? It's true. Let's say we wanted to find the area of the shaded shape in Figure 10.24. This could be any shape, possibly one whose area can't be exactly determined mathematically. But, we're smart chicken-farming programmers, so we know how to find an approximate value for the area.

We walk into our chickenyard and draw the shape on the ground, then go away and let the chickens walk about, pecking at the ground. We watch from a distance and count how many times they peck anywhere in the yard, and keep a separate count of the number of times they peck inside the shape we've drawn.

Now we're all set to estimate the area of the weird shape we've drawn. We know the dimensions of our chickenyard, and can calculate its area. We know the total number of pecks in the chickenyard, and we know how many of those pecks were inside the weird shape. To keep things clear, let's define some variables to represent these things:

$$\begin{aligned}
A_{total} &= \text{The total area of the chickenyard}\\
n_{total} &= \text{The total number of pecks}\\
n_{shape} &= \text{The number of pecks inside the shape}\\
A_{shape} &= \text{The unknown area of the shape}
\end{aligned}$$

If the total number of pecks is large and evenly spread through the chickenyard we'd expect that the following will be true:

$$\frac{A_{shape}}{A_{total}} = \frac{n_{shape}}{n_{total}}$$

Or, rearranging a little, we could say that

$$A_{shape} = A_{total} \frac{n_{shape}}{n_{total}} \tag{10.1}$$

We know all of the things on the right-hand side of this equation, so we just need to plug them into a calculator to find A_{shape}.
Thanks Chickens!

Fortunately for those who live in apartments, we don't need chickens in order to use this technique. We can do the same thing with a computer program.

As we've seen before, we can use the `rand` function to generate random numbers. Instead of letting chickens peck, we can write a program that generates pecks at random locations. Beyond that, the only thing our program will need will be some way of knowing whether a particular point lies inside the shape we're interested in.

Let's try doing it with a shape whose area we know exactly, so we can see if our pecking technique really does do a good job of estimating the area. Figure 10.25 shows a circular area inside a square chickenyard. The circle has a radius of 1, and we know that the area of a circle is πr^2, so the area of this circle should be exactly π.

"A balmy, restful peacefulness seemed to reign everywhere. Even the old hen seemed well satisfied. She scratched among the stones and called to her chickens when she found a treasure; and all the while clucked to herself with intense inward satisfaction. Waldo, as he sat with his knees drawn up to his chin and his arms folded on them, looked at it all and smiled. An evil world, a deceitful, treacherous, mirage-like world, it might be; but a lovely world for all that, and to sit there, gloating in the sunlight, was perfect. It was worth having been a little child, and having cried and prayed, so one might sit there."
—*The Story of an African Farm*, Olive Schreiner (1883).

Source: Wikimedia Commons

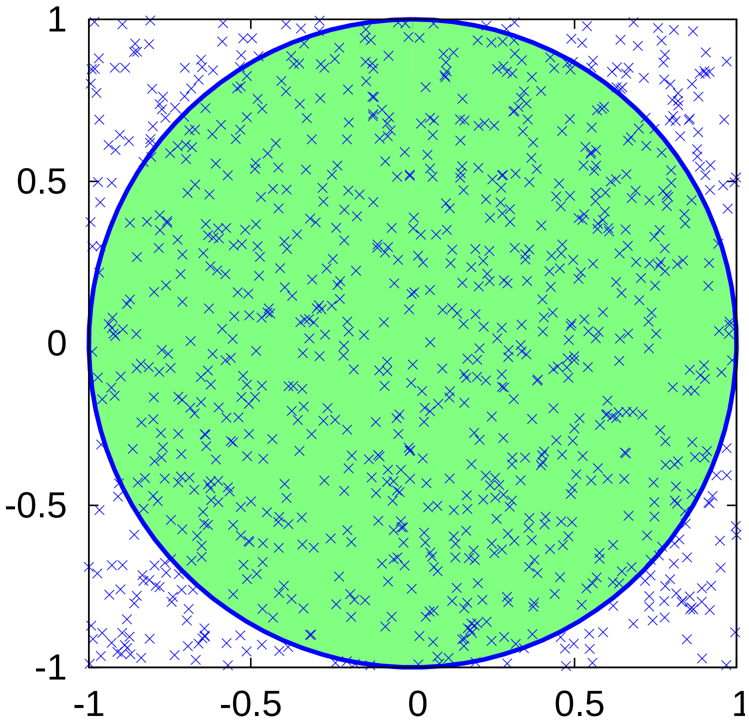


Figure 10.25: A circular area with a radius of 1, inside a square 1 × 1 "chickenyard".

Program 10.7 estimates the area of the circle by simulating chicken pecks. It generates 1,000 random peck positions inside the chicken yard, then checks each peck to see if it's inside our area of interest. The program keeps track of the number of pecks inside the area. After it's done pecking, the program calculates the area by using Equation 10.1, above.

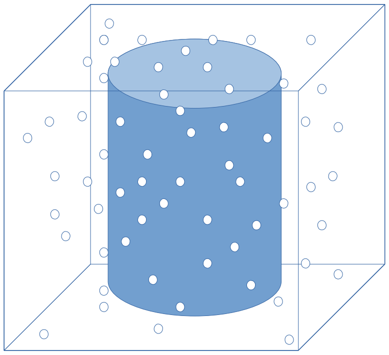


Figure 10.26: A 3-d version of our pecking scheme.

Note that each random peck position is created by generating random values for x and y. We do this by multiplying the width or height of the chickenyard by a random number between 0 and 1, and then adding the result to the minimum x or y value.

The function inside tells the program whether a given peck lies inside the area we're interested in. it checks to see if $\sqrt{x^2 + y^2}$ is less than or equal to the circle's radius. If the peck is inside, the function returns the value "1". Otherwise, it returns a "0".

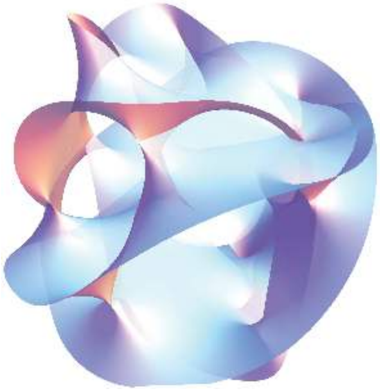
The inside function is the only part of the program that's specific to a circle. If we wanted to find the area of a different shape, we'd only need to rewrite this function.

The program could be extended to three dimensions by adding a z coordinate, and used to estimate volumes. In that case, we'd generate random x, y, and z coordinate for each "peck", and keep track of how many landed inside a 3-d shape we were interested in (see Figure 10.26).

The Monte Carlo Casino, in the Principality of Monaco

Source: *Wikimedia Commons*

The technique described in this section is known as "Monte Carlo integration". It takes its name from the gambling resort of Monte Carlo, on the French Riviera. Like the dice-rolling gamblers at Monte Carlo, our pecking program does its job by generating random numbers.



Consider the case of an 11-dimensional integral in String Theory. This might be utterly impossible to solve exactly, but by generating thousands of sets of 11 coordinate values, we could estimate its value by Monte Carlo methods.

Source: *Wikimedia Commons*

The Monte Carlo technique has several virtues:

- As we mentioned above, the Monte Carlo method can easily be extended to higher dimensional problems.
- To use this technique you only need two things:
  - You need a way to determine whether a point is inside the shape you're interested in, and
  - You need to be able to draw a "chickenyard" of a known area that completely encloses the shape.
- Although other methods are often more efficient for integrating 2-d functions, the Monte Carlo technique is quite efficient for higher-dimensional integrals.

Program 10.7: peck.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

int inside ( double x, double y ) {
  double radius = 1.0;
  if ( sqrt( x*x + y*y ) <= radius ) {
    return ( 1 );
  } else {
    return ( 0 );
  }
}

int main () {
  double xmin = -1, xmax = 1;
  double ymin = -1, ymax = 1;
  double atotal;
  double ashape;
  double xrange, yrange, x, y;

  int ntotal = 1000;
  int peck;
  int nshape=0;

  xrange = xmax - xmin;
  yrange = ymax - ymin;

  srand(time(NULL));

  for ( peck=0; peck<ntotal; peck++ ) {
    x = xmin + xrange*rand()/(1.0+RAND_MAX);
    y = ymin + yrange*rand()/(1.0+RAND_MAX);

    if ( inside( x, y ) ) {
      nshape++;
    }
  }

  atotal = (xmax-xmin) * (ymax-ymin);
  ashape = atotal * nshape/ntotal;

  printf ( "The area of the shape is %lf\n", ashape );
}
```

Exercise 54: Chicken Pot Pi

Create, compile, and run Program 10.7. Does it give a good approximation of the value of π? Try increasing the number of pecks to 100,000. Does this improve the program's results?

Now try integrating the `sin(sin(x))` function again, this time using the Monte Carlo technique. To do this, make a modified version of Program 10.7 as follows:

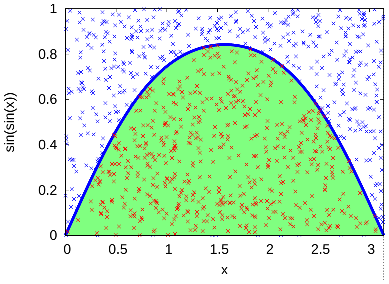


Figure 10.27: Integrating $sin(sin(x))$ using the Monte Carlo method.

- Copy `peck.cpp` to `peck2.cpp`:

```
cp peck.cpp peck2.cpp
```

- Change the `inside` function so it looks like this:

```
int inside ( double x, double y ) {
  if ( y <= sin(sin(x)) ) {
    return ( 1 );
  } else {
    return ( 0 );
  }
}
```

- Change the limits of the "chickenyard" to these values:

```
  double xmin = 0, xmax = M_PI;
  double ymin = 0, ymax = 1;
```

Compile and run your new `peck2.cpp` program. Does its result agree with the value you got earlier using the trapezoid rule to integrate `sin(sin(x))`?

$sin(x)dx = 2$

Figure 10.28: A chickintegral

10.12. Conclusion

The trapezoid rule and Monte Carlo integration techniques are both useful tools for dealing with recalcitrant integrals. They let a programmer find approximate values for the area under a curve defined by data points, and find arbitrarily precise approximations to a wide array of definite integrals of functions, many of which can't be solved exactly.

As Mick Jagger says, "You can't always get what you want, but if you try sometimes you just might find you get what you need."

Source: Wikimedia Commons

Practice Problems

1. Copy Program 10.4 into a new program named `cosarea.cpp` and modify it so that it estimates the integral of the cosine function between $x = 0$ and $x = \pi/4$ using ten slices. (See Figure 10.29.) The answer should be about 0.7.

2. Figure 10.30 shows the shape of the Rio Grande river bed near Bernalillo, New Mexico. Researchers measured the depth of the water at various positions as they waded across the river from one bank to the other. The resulting graph shows what a slice through the river would look like. We can use this data to estimate the cross-sectional area of the river. It would just be the shaded area shown in Figure 10.30, which is the area between the water's surface and the streambed. The data collected by the researchers is shown in Figure 10.31.

   Using Program 10.3 as a starting point, write a program named `riogrande.cpp` that finds the river's cross-sectional area. The program should read a data file named `rio-grande.dat` that contains the data in Figure 10.31. At the end, the program should print an estimate of the area, in square feet. (Note that the answer will be negative since all of the heights are below the water surface. You can just ignore the minus sign.)

   Your program should come up with an area of about 480 square feet. If we know this area, and we measure how fast the water is flowing (by dropping in a leaf and timing it between two points, for example) we can calculate the flow rate of water through the river. For example, if the water is flowing at 2 feet per second, that would mean that the flow rate is 480 × 2 = 960 cubic feet per second. That's over 7,000 gallons per second!

The Rio Grande near Bernalillo, New Mexico.*Source: Wikimedia Commons*

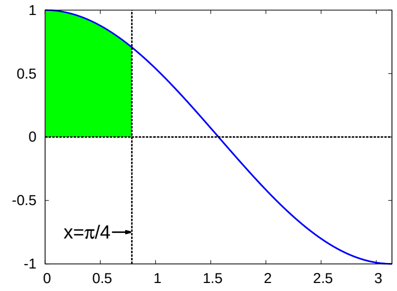


Figure 10.29: A graph of the cosine function. The region between $x = 0$ and $x = \pi/4$ is shaded.

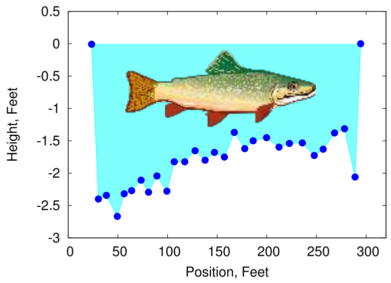


Figure 10.30: A cross-section of the Rio Grande near Bernalillo, NM. (Trout not to scale.)
Data from: J.E. Veenhuis, 2002, Summary of Flow Loss between Selected Cross Sections on the Rio Grande in and near Albuquerque, New Mexico: U.S. Geological Survey Water-Resources Investigations Report 02-4131.

```
24 -0.0
30 -2.4
38 -2.3
49 -2.7
56 -2.3
64 -2.3
73 -2.1
81 -2.3
89 -2.0
99 -2.3
107 -1.8
117 -1.8
128 -1.7
138 -1.8
147 -1.7
157 -1.8
167 -1.4
178 -1.6
186 -1.5
200 -1.5
212 -1.6
223 -1.5
236 -1.5
248 -1.7
257 -1.6
268 -1.4
278 -1.3
289 -2.1
294 0.0
```

Figure 10.31: Rio Grande cross-section data. The first column is distance and the second column is water depth, both measured in feet.

3. The Lorentzian function is often used to describe the shape of lines in a spectrum. It can be written as:

$$\frac{1}{\pi\gamma\left[1+\left(\frac{x-x_0}{\gamma}\right)^2\right]}$$

where x_0 and γ are parameters that control the shape of the function. Figure 10.32 shows a Lorentzian function with $x_0 = 0$ and $\gamma = 2$, over the range $x = -10$ to $x = 10$, divided into five or six slices.

Write a program named lorentzian.cpp that estimates the area under this curve. Look at Program 10.4 for inspiration. The program should ask you (using scanf) how many slices you want to use, and it should print the estimated area when it's done.

If you start out with a small number of slices (two, for example) and try the program several times with increasing numbers of slices, you should see that the estimates of the area jump up and down at first, then settle into a value of around 0.87. Figure 10.32 shows why this happens. For a small number of slices, odd numbers tend to underestimate the curve's area, and even numbers tend to overestimate it.

(**Note:** The symbol γ is the Greek letter *gamma*, but you'll run into trouble if you try to have a variable named gamma in your program. This is because C's math library contains a function named gamma. If you give a variable the same name, the compiler will get confused. Instead, you might use g to represent γ in your program.)

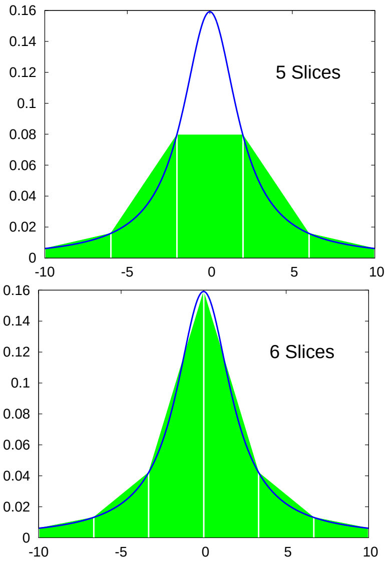


Figure 10.32: A Lorentzian function divided into five and six slices.

4. Program 10.7 (peck.cpp) estimates the area of a circle using the Monte Carlo method. Copy this program into a new file named sphere-peck.cpp and modify the new program so that it estimates the volume of a sphere.

To do this, you'll need to add a z dimension to go along with the x and y dimensions that are in the old program. That means adding variables named z, zmin, zmax, zrange.

The volume of the "chickenyard" will be the box surrounding the sphere, as shown in Figure 10.33. The volume of a box is just its height × width × depth.

You'll need to modify your inside function so that it takes three arguments instead of just two. The distance of a "peck' from the origin will be $\sqrt{x^2+y^2+z^2}$.

The real volume of the sphere is $\frac{4}{3}\pi$ or approxmiately 4.19. Your program should come up with a value similar to this if you use a thousand or more pecks.

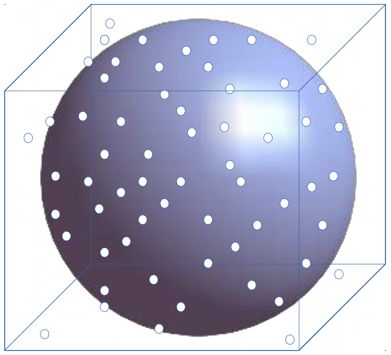


Figure 10.33: A sphere in a box. The "pecks" are represented by dots.

5. Find the approximate area under the curve $y = 1 - x^2$ between $x = -1$ and $x = 1$ (see Figure 10.34) using two different techniques:

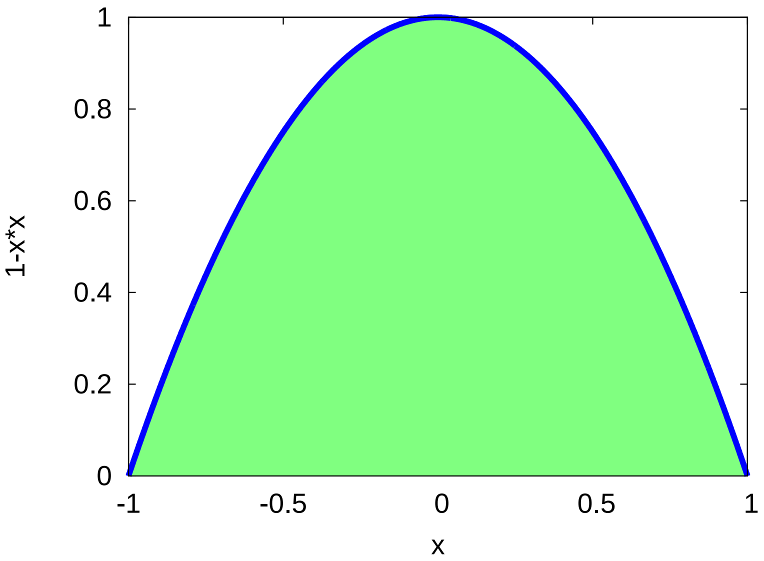


Figure 10.34: A parabolic area between $x = -1$ and $x = 1$.

(a) Copy Program 10.4 into a new file named `parabola.cpp`. Modify the new program so that, instead of finding the integral of the sine function, it finds the integral of $1 - x^2$ over the range from $x = -1$ to $x = 1$.

To do this, you'll need to change the `func` function and you'll need to change the values of `xmin` and `xmax`.

Compile and run your program. It should find that the area is approximately $\frac{4}{3}$. If the result you get initially isn't very close, try increasing the value of `nsteps` (the number of slices).

(b) Copy Program 10.7 into a new file named `parabola-peck.cpp`. Modify the program so that it finds the integral of $1 - x^2$ over the range from $x = -1$ to $x = 1$.

To do this, you'll need to modify the `inside` function. The function should return a 1 for points in the shaded region of Figure 10.34 and zero otherwise. (**Hint:** Check to see if `y < 1-x*x`.) You'll also need to change the bounds of your chickenyard, since `ymin` should now be zero.

Compile and run your program. It should find that the area is approximately $\frac{4}{3}$. If the result you get initially isn't very close, try increasing the value of `ntotal` (the number of pecks).

6. Find the approximate area under the curve $y = |sin(x)|$ (the absolute value of $sin(x)$) between $x = 0$ and $x = 2\pi$ (see Figure 10.35) using two different techniques:

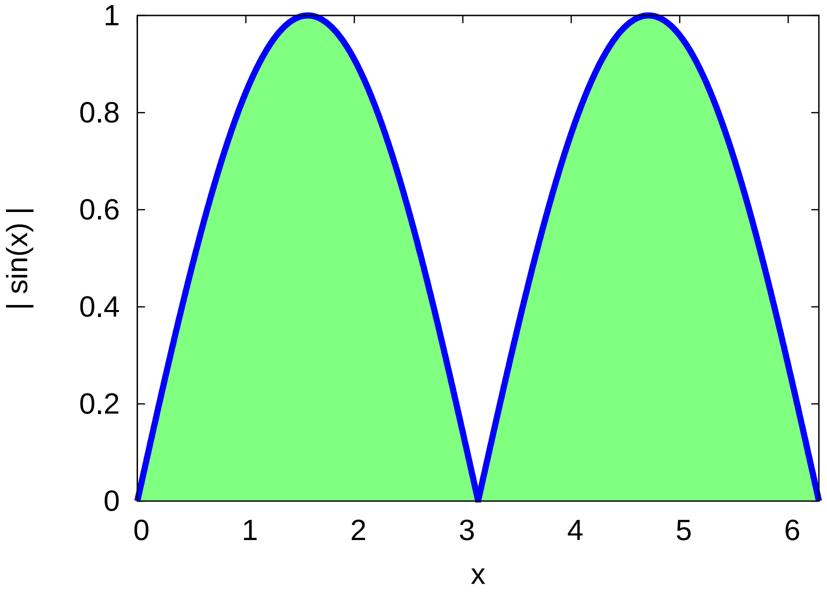


Figure 10.35: $|sin(x)|$ between $x = 0$ and $x = 2\pi$.

(a) Copy Program 10.4 into a new file named `fabsine.cpp`. Modify the new program so that, instead of finding the integral of the sine function, it finds the integral of `fabs( sin(x) )` over the range from $x = 0$ to $x = 2\pi$.

To do this, you'll need to change the `func` function and you'll need to change the values of `xmin` and `xmax`. Set the `nsteps` to 100.

Compile and run your program. It should find that the area is approximately 4. If the result you get initially isn't very close, try increasing the value of `nsteps` (the number of slices).

(b) Copy Program 10.7 into a new file named `fabsine-peck.cpp`. Modify the program so that it finds the integral of `fabs(sin(x))` over the range from $x = 0$ to $x = 2\pi$.

To do this, you'll need to modify the `inside` function. The function should return a 1 for points in the shaded region of Figure 10.35 and zero otherwise. (**Hint:** Check to see if `y < fabs( sin(x) )`.) You'll also need to change the bounds of your chickenyard, since `ymin` should now be zero. Set `ntotal` to 10,000.

Compile and run your program. It should find that the area is approximately 4. If the result you get initially isn't very close, try increasing the value of `ntotal` (the number of pecks).

7. There's a set of special functions named "Bessel functions" that are used in describing the shape of a vibrating drumhead (see Figure 10.36). In C, the first Bessel function is called `j0`. Find the approximate area under the curve $y = j0(x) + 1$ between $x = 0$ and $x = 16$ (see Figure 10.37) using two different techniques, while looking to see which technique gives a good answer more quickly:

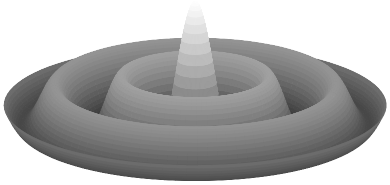


Figure 10.36: The shape of a vibrating drumhead can be described with the help of Bessel functions.

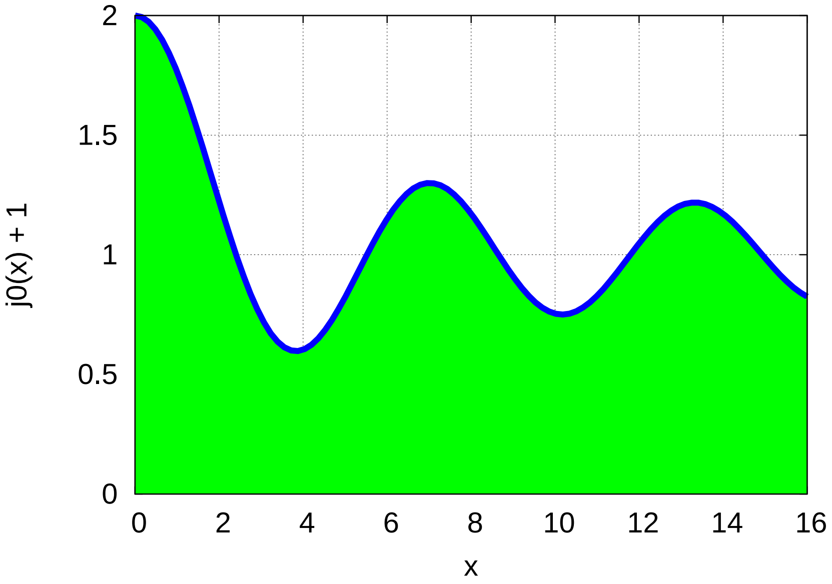


Figure 10.37: The function $j0(x) + 1$ between $x = 0$ and $x = 16$.

(a) Copy Program 10.4 into a new file named `bessel.cpp`. Modify the new program so that, instead of finding the integral of the sine function, it finds the integral of $j0(x) + 1$ over the range from $x = 0$ to $x = 16$.

Instead of setting `nsteps` equal to 5, modify the program so that it tries many different values of `nsteps`. Make a loop that tries each value of `nsteps` between 1 and 1,000. For each value, print `nsteps` and the estimated area. The result should be two columns of numbers, separated by a space.

Remember to reset your estimate of the area to zero whenever you change to a new value of `nsteps`. You'll need to modify the `func` function and you'll need to change the values of `xmin` and `xmax`.

Compile and run your program. When `nsteps` is large, you should find that the area is approximately 17.1. Notice that the program's estimate of the area is very different from this when `nsteps` is small, but it rapidly appraches the correct value.

(b) Copy Program 10.7 into a new file named `bessel-peck.cpp`. Modify the program so that it finds the integral of $j0(x) + 1$ over the range from $x = 0$ to $x = 16$.

Instead of setting `ntotal` to 1,000, try different values of `ntotal`. Make a loop that tries each value of `ntotal` between 1 and 1,000. For each value, print `ntotal` and the estimated area. The result should be two columns of numbers, separated by a space.

Remember to reset `nshape` to zero whenever you change the value of `ntotal`. You'll need to modify the `inside` function. The function should return a 1 for points in the shaded region of Figure 10.37 and zero otherwise. (**Hint:** Check to see if `y<j0(x)+1`.) You'll also need to change the bounds of your chickenyard, since `xmin` and `ymin` should now be zero, and `xmax` should be 16.

Compile and run your program. It should find that the area is approximately 17.1. Notice that the program eventually gets close to this value, but not until `ntotal` is pretty large.

If we ran our two programs like this:

```
./bessel > bessel.dat
./bessel-peck > bessel-peck.dat
```

we could compare the two output files by plotting them with *gnuplot*. The result is shown in Figure 10.38. The vertical axis shows our estimate of the area and the horizontal axis shows how many slices or pecks we needed to get that estimate. As you can see, the Monte Carlo program `bessel-peck.cpp` eventually gets close to the right answer, but it takes many "pecks" to get there. The other program (`bessel.cpp`) gets close to the right answer after only a few trapezoidal slices, and stays there from then on.

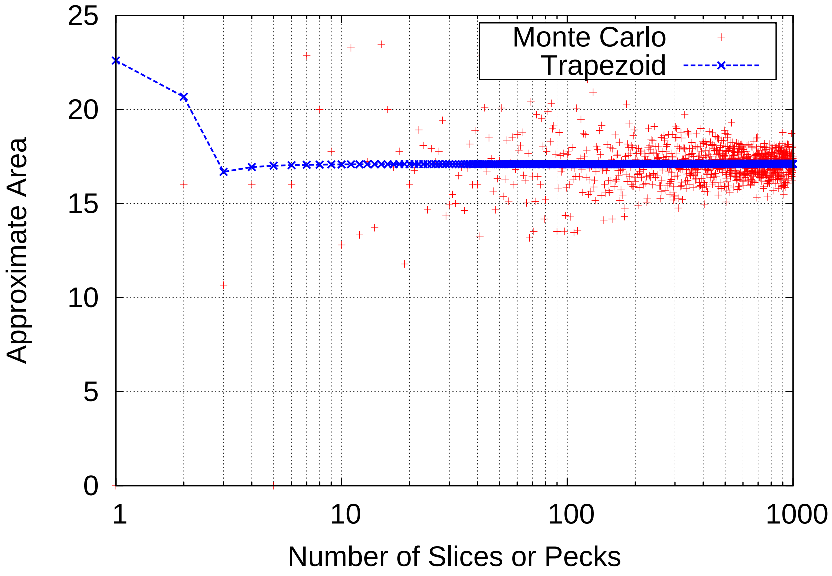


Figure 10.38: A comparison of area estimates for different values of `nsteps` or `ntotal`. Notice that the x axis is logarithmic, so we can pack a large range into it while still being able to see the small values. In *gnuplot* you can do this by saying `set log x` before you use the `plot` command.

8. Consider the graph displayed in Figure 10.39. It shows two parabolas, with the area between them shaded. The parabolas are described by these two equations:

$y = 1 - x^2$	Top parabola
$y = x^2 - 1$	Bottom parabola

   (a) Can you write a program that estimates this area using the Trapezoid Rule? Call the program `2parabola.cpp`. You might look at Program 10.4 as an example to get you started.

   (b) Can you write a program that estimates the area using Monte Carlo methods? Call this program `2parabola-peck.cpp`. Look at Program 10.7 for inspiration.

   Both programs should find that the area is about $\frac{8}{3}$ ($\simeq 2.67$).

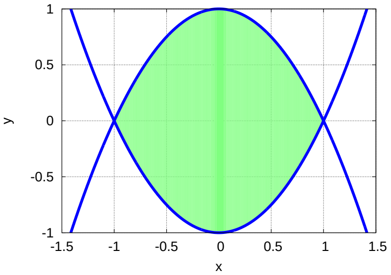


Figure 10.39: The intersection of two parabolas.

9. Copy Program 10.4 into a new program named `atanarea.cpp` and modify it so that it estimates the integral of the arctangent function (the `atan` function in C) between $x = 0$ and $x = \pi$ using ten slices. (See Figure 10.40.) The answer should be about 2.77.

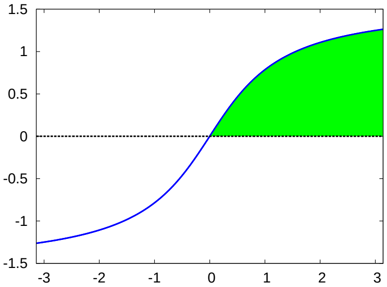
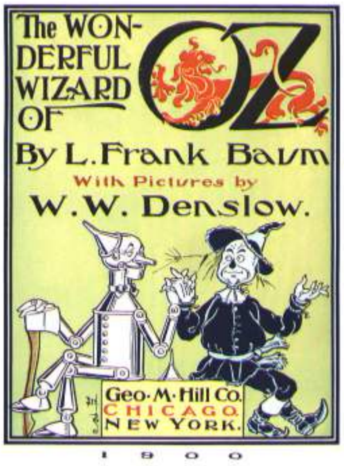


Figure 10.40: A graph of the arctangent function. The region between $x = 0$ and $x = \pi$ is shaded.

11. Libraries

11.1. Introduction

> *"Pay no attention to the man behind the curtain!"*
> *—The Wizard of Oz*



The Wonderful Wizard of Oz, by L. Frank Baum (1900). In 1902 it was made into a Broadway musical, and a film in 1939.

Source: Wikimedia Commons

Ah, but we must, dear Wizard. The time has come to lift the veil that's hidden some of C's inner workings. In particular, we'll now take a look at the place where the C compiler finds all of those functions we've been using: things like `printf`, `sqrt`, and `rand`. We've seen that we can write our own functions, but where do these "built-in" functions live. Somewhere over the rainbow?

As we'll see, these functions are collected in "libraries". This kind of library contains pre-compiled snippets of code that *g++* can plug into your programs.

Some programmer long ago wrote a function called `sqrt`, just as you've written functions in your own programs. This function was then converted into binary instructions that a computer can understand, and stored in a library for later use. When *g++* compiles a program that uses the `sqrt` function, it finds this chunk of binary instructions in the library and inserts it into your program.

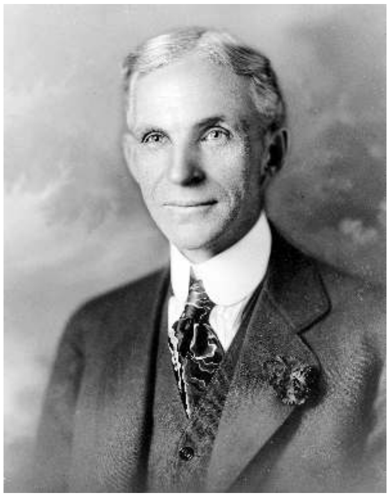
Source: Wikimedia Commons

If you write a program that uses a function named `flyingmonkeyspeed`, *g++* first looks at your `cpp` file to see if you've written your own function with that name. If not, *g++* then looks through a standard list of libraries to see if one of them contains a function with that name. If no function is found in either place, *g++* gives you an error message.

In this chapter we'll explore libraries and other strange creatures related to the inner workings of the *g++* compiler. Let's take a stroll down the Yellow Brick road and see what we find.

11.2. The g++ Assembly Line

L. Frank Baum's *The Wonderful Wizard of Oz*, published in 1900, was an American fairy tale that celebrated the ingenuity and inventiveness that was in the air at that time. Orville and Wilbur Wright were making manned glider flights at Kitty Hawk. The Automobile Club of America held the first automobile race in the United States. Henry Ford would found the Ford Motor Company three years later, based on revolutionary principles of assembly line production.



The dark side of Henry Ford: Ford was an outspoken antisemite, and mandated the distribution of a copy of the rabidly antisemitic newspaper, *The Dearborn Independent*, with each Ford automobile sold. In Germany, Nazi leaders cited Ford's influence on their movement. Ford is mentioned favorably in Hitler's *Mein Kampf* and, in a 1931 interview, Hitler said that Ford was his "inspiration".

Source: Wikimedia Commons

Ford's assembly lines are a good analogy for what we'll be talking about in this chapter. To understand why, we'll need to look inside *g++*.

We've learned that *g++* takes a line like this:

```
        printf ("Hello, world!\n");
```

and translates it into instructions the computer can understand, like this:

```
        100111000101011101111000100111100l.......
```

g++ actually does this job in three discrete steps, called "preprocessing", "compiling", and "linking", and interesting things happen at each stage.

When you type `g++ -Wall -o hello hello.cpp` you can imagine your program travelling along an assembly line. At each stop along the assembly line, the program is modified or translated in some way, until a shiny new binary program pops out at the far end, ready to be run.

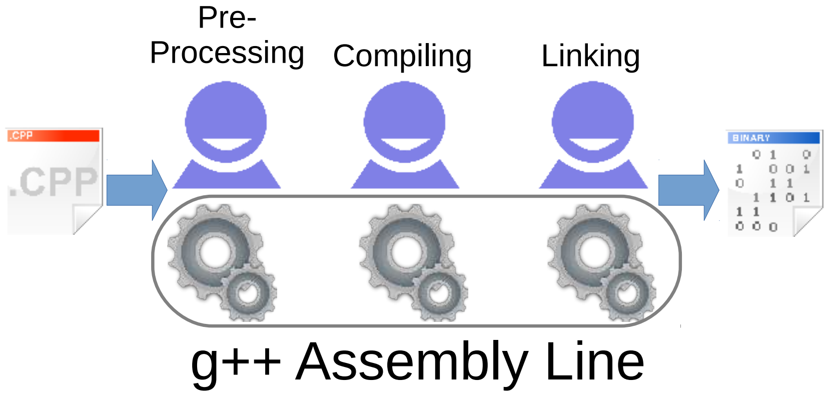


Figure 11.1: g++ does its work in several stages.

So far, we've only talked about the middle step, where *g++* translates C language statements into binary. Now let's look at the other two steps, starting with "preprocessing".

11.3. Preprocessing

You might be surprised to learn that the #include statements we've been putting at the top of our programs aren't really part of the C language at all. Instead, they belong to a separate "C preprocessor language". All of the statements in this language begin with #. The C preprocessor provides you with some handy shortcuts that make writing C programs easier. The most useful of these is #include.

When you compile a program, *g++* begins by running your file through the preprocessor. When the preprocessor sees #include <stdio.h> it searches through a predefined list of directories[1], looking for a file named stdio.h. When it finds the file, the preprocessor inserts this file's contents into your program just as though you had typed them directly in at the spot where you said #include <stdio.h>. If stdio.h can't be found (maybe you typed its name wrong?) you'll get an error message.

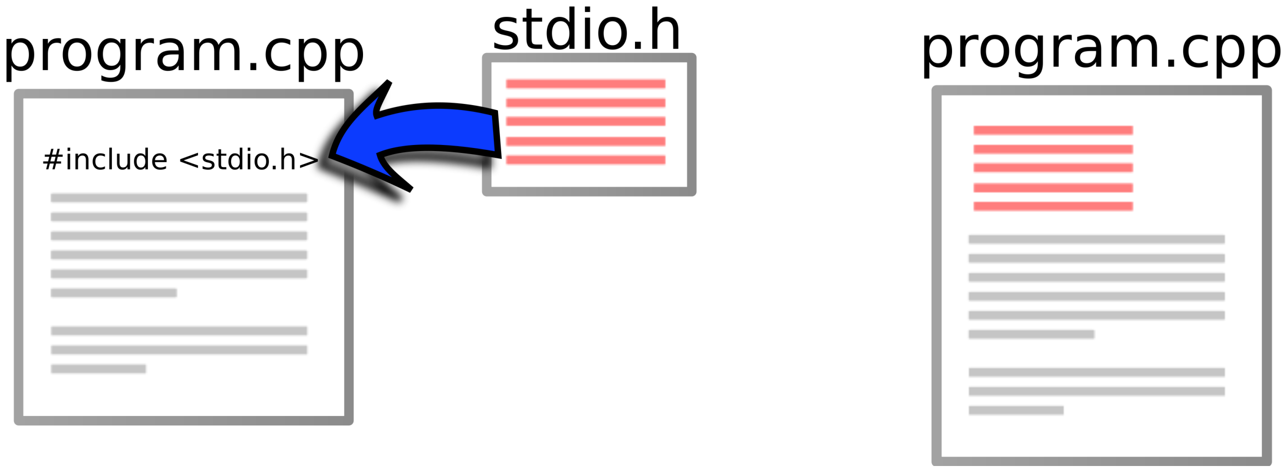


Figure 11.2: The C preprocessor takes the contents of stdio.h and inserts them into your program.

The ".h" in the name of files like stdio.h stands for "header". The content of these files, when #include'ed, acts as a header at the top of your program that defines some symbols (like M_PI from math.h), or prepares your program to use some functions.

There are a couple of variations on the #include statement, and how they behave might vary slightly from one C compiler to another. In general:

[1] Remember that "directory" is just another word for "folder".

- `#include "file.h"`, with quotes around the file name, first looks for file.h in the same directory as the program, then searches through the predefined list of directories.

- `#include <file.h>`, with angle brackets[2] around the file name, only looks through the predefined list of directories.

As you've probably guessed by now, you can write your own header files to be included in your program. If you do this, best practice is to use `#include "file.h"` for your own files, and reserve `#include <file.h>` for system files.

But what about...?

Can you find out where *g++* will look for files like `stdio.h`?

The list of directories that are searched will vary from one kind of computer to another, but you can see the list by typing the following magic command, which invokes the preprocessor (named `cpp`) directly:

```
echo | cpp -xc++ -Wp,-v -P
```

The output should look something like this:

```
#include "..." search starts here:
#include <...> search starts here:
  /some/directory/some/where
  /some/other/directory
  /maybe/another/directory
End of search list.
```

If you want to spy on what a program looks like after being run through the preprocessor, type "`cpp -P hello.c > hello.out`" and look at `hello.out` with *nano*. Near the bottom of the file you'll see the C statements from your original program, but most of the file will be the contents of `stdio.h`.

11.4. Some Handy Random-Number Functions

Source: Wikimedia Commons

Let's look in on Dorothy and see how she's progressing down the Yellow Brick Road. Hmm. It looks like she's still in Munchkinland. Those pesky little Munchkins are swarming around her, dancing and singing and generally getting underfoot. Sheesh! How's she ever supposed to make it to the Emerald City? And the Wicked Witch is looking for her, too. That swarm of Munchkins is like a big, neon,

[2] Also known as "less than" and "greater than" symbols.

"Come and Get Me!" sign.

Oh well. Since we're programmers, this whole situation just begs to be simulated. Let's try to make a model of the Munchkin distribution around Dorothy.

We'll probably need some random numbers to do that. Until now, we've been using the rand function directly, but we know how to write our own functions now, so let's write one that makes it easier to generate one sort of random numbers we're often interested in. Take a look at the function rand01 below.

```
double rand01 () {
  static int needsrand = 1;
  if ( needsrand == 1 ) {
    srand(time(NULL));
    needsrand = 0;
  }
  return ( rand()/(1.0+RAND_MAX) );
}
```

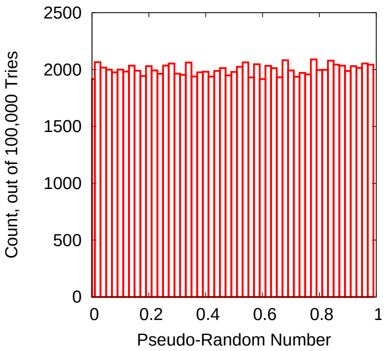


Figure 11.3: Histogram of 100,000 pseudo-random numbers generated by rand01.

The function rand01 generates a pseudo-random real number between zero and one (see Figure 11.3). The most important part of the function is just a return statement that sends back the value rand()/(1.0+RAND_MAX). We've used this in lots of programs already, but it's much easier to type rand01 than "rand()/(1.0+RAND_MAX)".

The function also saves us work in another way. Remember how we used the srand function to initialize the pseudo-random number generator so we get a different set of numbers each time we run the program? The rand01 function takes care of that for us.

To make sure it only uses srand once, the function defines a variable called "needsrand" ("need srand") that starts out with a value of 1. The first time rand01 is used, it invokes srand and then sets needsrand to zero. The next time rand01 is used it checks the value of needsrand and discovers that it doesn't need to use srand again.

Notice that needsrand is defined as "static". As we discussed in Chapter 9, variables inside functions are wiped out when the function finishes unless we declare them static. Since we want to use needsrand to remember what we did the last time rand01 was used, this variable needs to be static.

Let's assume that the Munchkins are swarming around Dorothy, each trying to get as close to her as possible, and elbowing each other out of the way occasionally. We might assume that the density of Munchkins would be highest near Dorothy, and fall off like a Normal curve at larger distances from her.

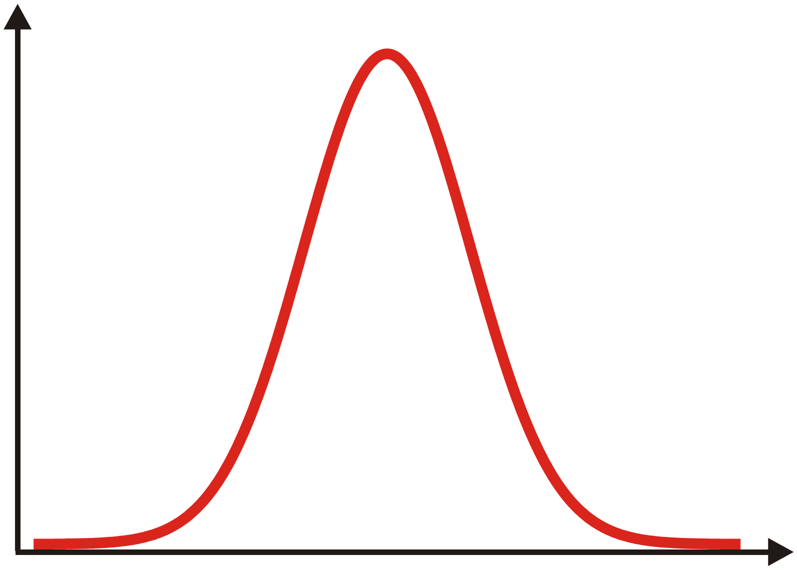


Figure 11.4: A Normal (bell-shaped) distribution. It resembles a slightly-melted witch's hat.

How can we generate pseudo-random numbers distributed like this? It turns out that there's a handy statistical trick for generating numbers in an approximately Normal distribution[3]. All we need to do is take 12 numbers generated by `rand01`, add them up, and subtract 6. The numbers obtained this way will be distributed approximately like a Normal distribution with a mean value of 0 and a standard deviation of 1 (see Figure 11.5.). That's what this function named `normal` does:

```
double normal () {
  int nroll = 12;
  double sum = 0;
  int i;
  for ( i=0; i<nroll; i++ ) {
    sum += rand01();
  }
  return ( sum - 6.0 );
}
```

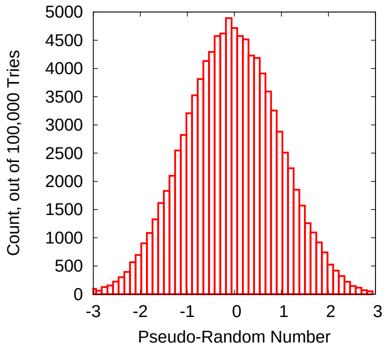


Figure 11.5: Histogram of 100,000 pseudo-random numbers generated by `normal`.

With those two functions, we're ready to simulate the distribution of Munchkins around Dorothy, as they might appear when viewed through the Wicked Witch's crystal ball. That's what Program 11.1 does.

[3] Why does this magic work? Unfortunately, that's beyond the scope of this book, but in general it relies on the Central Limit Theorem, mentioned in Chapter 7

Program 11.1: munchkin.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

const int nmunchkin=1000; // Munchkin population.

double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }
  return ( rand()/(1.0+RAND_MAX) );
}

double normal () {
  int nroll = 12;
  double sum = 0;
  int i;
  for ( i=0; i<nroll; i++ ) {
    sum += rand01();
  }
  return ( sum - 6.0 );
}

void xydump ( int npoints, double x[], double y[], char filename[] ) {
  FILE *output;
  int i;
  output = fopen( filename, "w" );
  for ( i=0; i<npoints; i++ ) {
    fprintf( output, "%lf %lf\n", x[i], y[i] );
  }
  fclose ( output );
}

int main () {
  int i;
  double x[nmunchkin], y[nmunchkin];
  double r, theta;
  char filename[] = "munchkin.dat";

  for ( i=0; i<nmunchkin; i++ ) {
    r = normal();
    theta = 2.0*M_PI*rand01();

    x[i] = r*cos(theta);
    y[i] = r*sin(theta);
  }
  xydump( nmunchkin, x, y, filename );
}
```

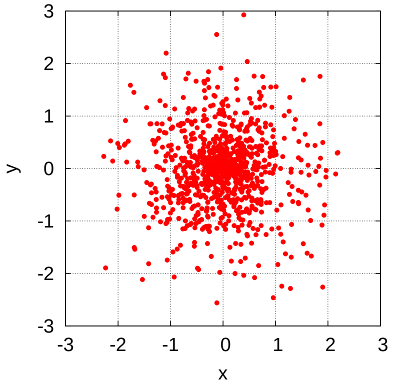


Figure 11.6: The view from the Witch's Munchkin-Scope.

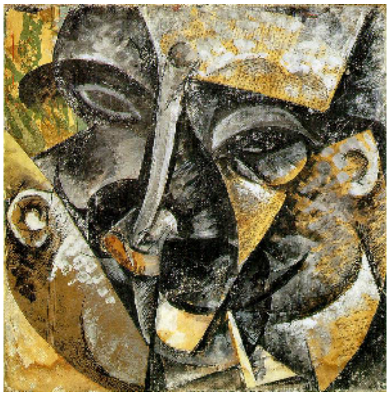
Notice that, for convenience, we've also created a function named `xydump` that writes the x and y coordinates of the Munchkin's positions into a file. When plotted with *gnuplot*, the result looks like Figure 11.6. (For this figure, I've turned on a grid by giving *gnuplot* the command "`set grid`".)

Program 11.1 gets the x and y coordinates by generating a random distance from Dorothy (`r`), with a Normal distribution centered on her, and a random angle (`theta`). A little trigonometry turns these numbers into the Cartesian coordinates x and y.

11.5. Making a Header File

Program 11.1 contains several functions that might be useful in other programs. We often need random numbers, and we often dump data into a file. We could always just copy the functions into the next program we write, but let's think about how we might make it easier to re-use these functions.

Take a look at Program 11.2. This program does the same thing as Program 11.1, but it's a lot shorter! That's because we've shoveled all of the functions (and our `nmunchkins` variable) into a new header file that we call `munchkin.h`.



Dynamism of a Man's Head, by Umberto Boccioni (1913).

Source: *Wikimedia Commons*

Speaking of heads, Thomas M. Disch's short story "Fun with Your New Head" is well worth reading.

Program 11.2: munchkin.cpp, with new header file

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#include "munchkin.h"

int main () {
  int i;
  double x[nmunchkin], y[nmunchkin];
  double r, theta;

  for ( i=0; i<nmunchkin; i++ ) {
    r = normal();
    theta = 2.0*M_PI*rand01();

    x[i] = r*cos(theta);
    y[i] = r*sin(theta);
  }
  xydump( nmunchkin, x, y, "munchkin.dat" );
}
```

Notice that we've used `#include "..."` instead of `#include <...>`, since this is a header file we've written ourselves (not a system file) and we'll keep it in the same directory where we keep `munchkin.cpp`. The file `munchkin.h` just contains the stuff we left out when we went from Program 11.1 to Program 11.2. It looks like this:

"I've got a header, but I'm still a no-brainer!"

Source: Wikimedia Commons

Program 11.3: munchkin.h

```
const int nmunchkin=1000; // Munchkin population.

double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }
  return ( rand()/(1.0+RAND_MAX) );
}

double normal () {
  int nroll = 12;
  double sum = 0;
  int i;
  for ( i=0; i<nroll; i++ ) {
    sum += rand01();
  }
  return ( sum - 6.0 );
}

void xydump( int npoints, double x[], double y[], char filename[] ) {
  FILE *output;
  int i;
  output = fopen( filename, "w" );
  for ( i=0; i<npoints; i++ ) {
    fprintf( output, "%lf %lf\n", x[i], y[i] );
  }
  fclose ( output );
}
```

We can compile Program 11.2 by typing `g++ -Wall -o munchkin munchkin.cpp`, just like any other program we've written. During the preprocessing phase, *g++* replaces "`#include "munchkin.h"`" with the contents of munchkin.h, and then proceeds just as though we'd typed those things directly into our program when we wrote it.

This is clearly one way that we could re-use our functions in another program. The next time we write a program that needs these functions, we can just add the line #include "munchkin.h" at the top and we'll have them.

Exercise 55: Munchkin Functions

Create the file "munchkin.h" (Program 11.3). Write a program named normaltest.cpp that uses #include "munchkin.h" to obtain the Munchkin functions we've written.

By using the normal function, have the program write out 10,000 pseudo-random numbers distributed in a Normal distribution.

Run the program like this:

```
./normaltest > normaltest.dat
```

then plot normal.dat using the *gnuplot* command plot "normal.dat". The result should look like Figure 11.7.

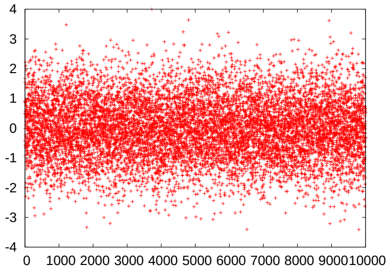


Figure 11.7: Exercise 55 should produce a graph like this.

11.6. Some Statistical Functions

Let's see how far Dorothy has gotten while we were simulating Munchkins.

Oh no! She's about to cross the poppy field! I wonder if she'll make it across without falling asleep? It looks like another simulation is called for.

When Baum wrote *The Wonderful Wizard of Oz* poppies brought to mind the soporific qualities of opium. By the time the book had become the 1939 film, poppies brought to mind the darker memories of World War I's Flanders Fields.

Source: Wikimedia Commons

Program 11.4 is the result. It simulates 1,000 runs through the poppy field. During each run, the time spent in the field is broken up into 1-minute segments. Using a "poppytoxicity" that tells us the probability of falling asleep after one minute's exposure to the poppies, and a random number (like rolling dice), the program tests to see if Dorothy fell asleep during each 1-minute segment. Every time she makes it all the way across the poppy field, the program increments a counter variable named nsuccess.

At the end, the program tells us the maximum distance covered in any run, the mean distance of all runs, and the standard deviation of the run distances.

The program uses the random-number functions from our previous program by `include`'ing `munchkin.h`. You'll notice that the program uses several variables that aren't visibly defined: `dorothyspeed`, `poppytoxicity`, and `poppyfieldsize`. At the bottom of the program there are also references to some new functions: `maxelement`, `mean`, and `stddev`. These missing things are all defined in a new header file named `poppy.h`.

The variables defined there look like this:

```
const double dorothyspeed = 4; // Dorothy's walking speed, mph
const double poppytoxicity = 0.5; // Prob. of sleep after one minute's exposure.
const double poppyfieldsize = 1.0; // Width of poppy field, in miles.
```

Notice that we've declared each of these variables (and `nmunchkin` in `munchkin.h`) to be "`const`". This tells the compiler that these numbers are constants, and shouldn't change. If we accidentally tried to change one of these values somewhere in our program, the compiler would give us an error message.

We also define some useful new functions in `poppy.h` (see "Program" 11.5). The first of these is `mean`, which tells us the mean value of an array of values. Similarly, the function `stddev` tells us the standard deviation of the values. These functions use techniques we talked about in Chapter 7. The last new function is `maxelement`, which finds the element number of the biggest value in an array. This is a function we've used already, in Program 9.14 in Chapter 9.

Zwei Schlafende Maedchen auf der Ofenbank, by the Swiss artist Albert Anker (1895).

Source: Wikimedia Commons

Poor Dorothy! With the running speed, toxicity, and field size we've given it, the program says she's very unlikely to make it across the field. On average, she would only make it about 6% of the way across, and even in the luckiest case she only gets $\frac{2}{3}$ of the way:

```
0 trials ended in success.
Max distance = 0.666667 miles
Mean distance = 0.061200 miles
Std. Dev. = 0.091728 miles
```

Program 11.4: poppy.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#include "munchkin.h"
#include "poppy.h"

int main () {

  double delta;
  double distance;
  double trial[1000];
  int i;
  int nsuccess = 0;

  delta = dorothyspeed/60.0; // Distance covered in 1 min.

  for ( i=0; i<1000; i++ ) {
    distance = 0;
    while (1) {
      if ( rand01() <= poppytoxicity ) {
        break;
      }
      distance += delta;
      if ( distance >= poppyfieldsize ) {
        nsuccess++;
        break;
      }
    }
    trial[i] = distance;
  }

  printf ("%d trials ended in success.\n", nsuccess );
  printf ( "Max distance = %lf miles\n", trial[ maxelement(1000,trial) ] );
  printf ( "Mean distance = %lf miles\n", mean(1000,trial) );
  printf ( "Std. Dev. = %lf miles\n", stddev(1000,trial) );
}
```

Program 11.5: poppy.h

```
const double dorothyspeed = 4; // Dorothy's walking speed, mph
const double poppytoxicity = 0.5; // Prob. of sleep after one minute's exposure.
const double poppyfieldsize = 1.0; // Width of poppy field, in miles.

double mean ( int nelements, double array[] ) {
  int i;
  double sum=0;
  for ( i=0; i<nelements; i++ ) {
    sum += array[i];
  }
  return ( sum/(double)nelements );
}

double stddev ( int nelements, double array[] ) {
  int i;
  double sum=0;
  double average;
  average = mean( nelements, array );
  for ( i=0; i<nelements; i++ ) {
    sum += pow(array[i]-average, 2);
  }
  return ( sqrt( sum/(nelements-1) ) );
}

int maxelement ( int nelements, double array[] ) {
  double max=0;
  int i, imax;
  for ( i=0; i<nelements; i++ ) {
    if ( array[i] > max ) {
      max = array[i];
      imax = i;
    }
  }
  return ( imax );
}
```

Exercise 56: Run Dorothy Run!

Create the files `poppy.cpp` (Program 11.4) and `poppy.h` (Program 11.5).

Compile and run the program to verify that your results match those obtained above. Then modify `poppy.h` by increasing Dorothy's speed. Re-compile the program and run it again. How fast does Dorothy need to run in order to have about a 50/50 chance of making it across? (In other words, in order to make it across successfully in 50% of the 1,000 trials.)

11.7. Some Histogram Functions

We can imagine that we might continue like this through our whole programming career, creating new functions and saving them in header files for later use. But what if we had thousands of functions, some of them long and complex. That's the case with C's collection of standard functions.

It could take *g++* several minutes to compile the contents of a very long header file, or a bunch of header files, containing thousands of functions. We don't want to wait that long to compile our program, especially if we only need one or two functions from our collection.

Let's try writing a new program, and use it as an opportunity to explore another way of saving functions for later use. What shall we write? We'll look to Dorothy again for inspiration.

Thanks to Glinda the Good, Dorothy has made it out of the poppy field, but now (gasp!) she's being chased by a swarm of flying monkeys.

The swarm contains some energetic young monkeys who always want to race ahead, and some lazy monkeys who always lag behind. When they start out chasing Dorothy they're all flying together, but after a mile or two they've spread out, with the fast flyers in front and the slower ones at the rear.

Let's write a program to make a histogram of the spatial distribution of the flying monkeys after they've flown for an hour. We'll need to use our random-number functions to set the speeds of the monkeys, and

For some reason, the town of Motala, Sweden, has on its coat of arms two flying monkeys and a propeller. This clearly deserves an explanation, but I can offer none.

Source: Wikimedia Commons

we can use our statistical functions to check the mean speed to make sure it looks reasonable. The result is Program 11.6.

The first thing you'll notice is that the program `include`'s the file `oz.h` instead of either of the header files we've written so far. Among other things, this file contains definitions for some new constants that we'll be using:

```
const int nmonkeys = 1000; // Number of flying monkeys in swarm.
const double meanmonkeyspeed = 25; // mph, same as an unladen European swallow.
const double monkeyspeedspread = 5; // mph, std. dev. of monkey speeds.
```

The program starts out by setting the speeds of the monkeys. It does this by starting with the mean monkey speed[4], then adding or subtracting some random amount based on our `normal` function. The program also initializes the position of each monkey to "0 miles" at this point.

In the program's second loop it steps through 60 minutes of time, minute by minute. In each "time slice" the program moves each monkey forward by an amount based on that monkey's speed.

After 60 minutes have passed, we make a histogram[5] of the monkeys' current positions. We start out by using a new function (which we'll see soon) named `resethist` to set all of this histogram bins to zero.

The program then loops through all of the monkeys and drops a "virtual marble" into the appropriate histogram bin for each, using another new function named `addtohist`. When it's all done with this, the program dumps the histogram data into a file, using our last new function `histdump`.

The output file (`monkey.dat`) will contain two columns: the distance travelled, and the number of monkeys that have travelled that distance. We could plot this with `gnuplot` and get a graph similar to Figure 11.8.

11.8. Linking

Okay, so that's a pretty picture (if you're into that kind of thing), but how did we get Program 11.6 to compile? Did we just pack all of our functions and constant definitions into the header file named `oz.h`?

No, we did something a little fancier. We created a library of Oz-related

[4] By this we mean mean *monkey speed*, not *mean monkey* speed, although the latter might be appropriate too.

[5] If you've forgotten how histograms work, take another look at Chapter 7.

Program 11.6: monkey.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#include "oz.h"

int main () {
  double speed[nmonkeys]; // Speed of each monkey.
  double position[nmonkeys]; // Total distance flown by each monkey.
  int minute;
  int monkey;
  double xmin, xmax;
  int nbins = 50;
  int bin[nbins];
  char filename[]="monkey.dat";

  for ( monkey=0; monkey<nmonkeys; monkey++ ) {
    speed[monkey] = meanmonkeyspeed + monkeyspeedspread*normal();
    if ( speed[monkey] < 0.0 ) {
      speed[monkey] = -speed[monkey]; // "Hey buddy, turn around!"
    }
    position[monkey] = 0.0;
  }

  printf ( "Min speed = %lf\n", speed[ minelement(nmonkeys,speed) ] );
  printf ( "Max speed = %lf\n", speed[ maxelement(nmonkeys,speed) ] );
  printf ( "Mean speed = %lf\n", mean( nmonkeys, speed ) );

  for ( minute=0; minute<60; minute++ ) {
    for ( monkey=0; monkey<nmonkeys; monkey++ ) {
      position[monkey] += speed[monkey]/60.0;
    }
  }

  resethist(nbins,bin);

  xmin = position[ minelement(nmonkeys,position) ];
  xmax = position[ maxelement(nmonkeys,position) ];

  for ( monkey=0; monkey<nmonkeys; monkey++ ) {
    addtohist( nbins, bin, xmin, xmax, position[monkey] );
  }
  histdump ( nbins, bin, xmin, xmax, filename );
}
```

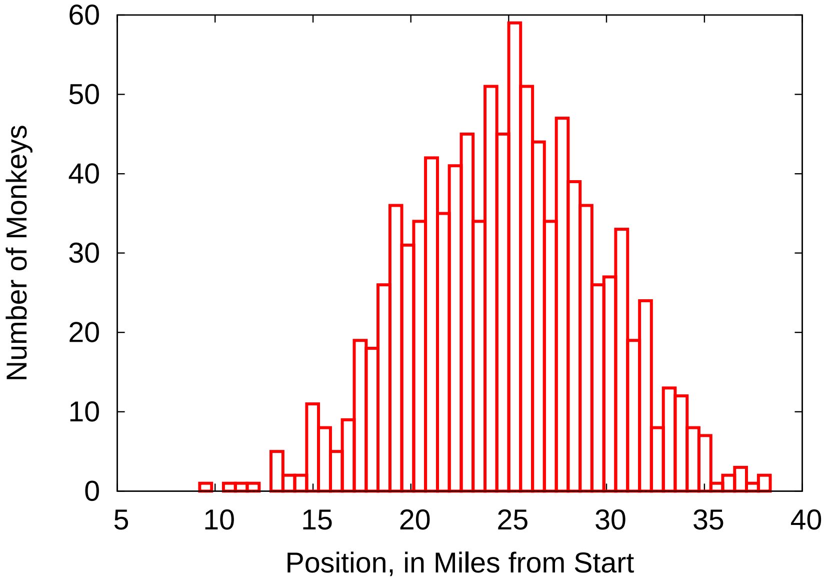


Figure 11.8: The distribution of the monkey swarm after flying for one hour.

functions.

Before talking about libraries, we need to return to the *g++* "assembly line". (Refer back to Figure 11.1.)

After preprocessing your `hello.cpp` file, the main work of the compiler happens. *g++* takes the preprocessed C code and converts it into a binary form that's digestible by the CPU. But what about functions that aren't defined in our program, like "`printf`"? How can the C compiler write CPU instructions for these functions? In fact, it can't: instead, it just inserts placeholders in the code for now.

Source: *Wikimedia Commons*

The placeholders referring to things that aren't in your hello.cpp file are resolved in the final step, which is called "linking". In this stage, *g++* invokes another program, called "`ld`", which looks through a set of standard libraries, trying to find a function called `printf`. We'll talk about how libraries are created soon, but for now you just need to know that a library contains pre-compiled chunks of code that correspond to functions like `printf`. The linker copies any chunks it needs from the libraries, and inserts them into the appropriate places in your program.

There are three important things to note about linking:

- First, the chunks of code in the libraries are pre-compiled, so they're already binary code that's ready to be used by your CPU.

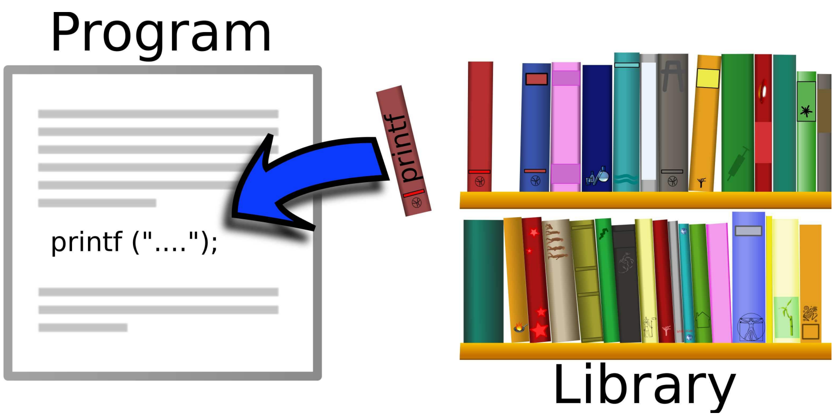
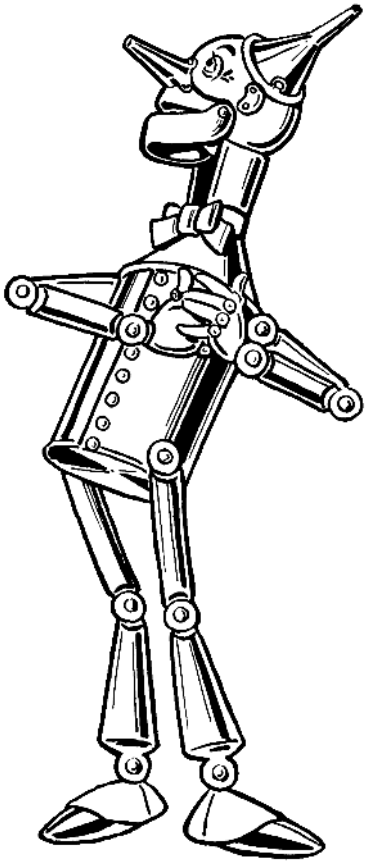


Figure 11.9: *g*++ looks through libraries to find any missing functions.

- Second, if the linker can't find a chunk of code corresponding to a function that you've used, it will spit out an error message telling you that it has run into an unresolved reference (your program refers to a function that can't be found). This may mean that you need to tell the compiler to look elsewhere, in other libraries besides the standard ones. (Or it may mean that you have a typo in your program!)
- Third, the linker only copies the functions that your program really uses. It doesn't insert a copy of the whole library into your program.



"I heart libraries!"

Source: Wikimedia Commons

11.9. Creating a Library

It's very easy to create a library of your own. Say, for example, that we have a file called oz.cpp that contains a lot of spiffy Oz-related functions that we've written. The file doesn't contain a complete program (there's no "`main()`"), it just contains the Oz functions. It might look like Program 11.7.

`oz.cpp` contains all of the Munchkin, poppy, and flying monkey functions that we've written so far in this chapter.

The first step in turning this into a library is to convert our C code into binary code. This isn't a whole program, so we're going to skip the "linking" step that g++ did in the example above. We can do this by typing:

```
g++ -Wall -c oz.cpp
```

This tells g++ to just do the pre-processor and compile steps and then stop. It produces an output file called oz.o, where the .o stands for "object". An object file contains binary code that has been compiled, and is ready to be inserted into a program.

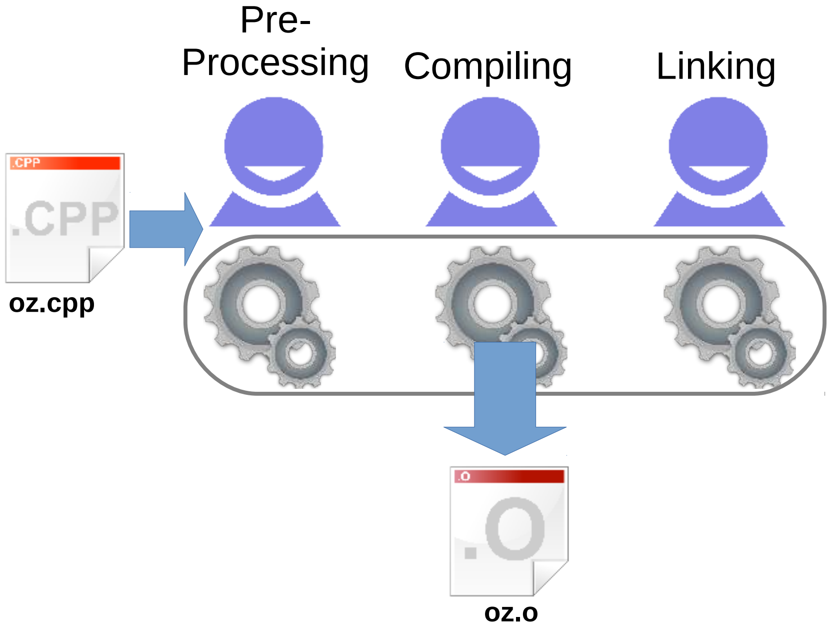


Figure 11.10: An "object" file is created by converting your C code into binary, but not plugging in any functions from libraries.

The "`ar`" command[6] can be used to pack object files into a library and index them for later use. For example, we could create a new library containing our Oz functions:

```
ar -csr liboz.a oz.o
```

where "c" means "create the library if it doesn't exist", "s" means "generate an index", and "r" means "replace anything of the same name that is already in the library".

By default, *g++* looks for functions like `printf` in a set of system libraries that are installed along with `g++`. A library named `libm.a` contains most of the math functions, and `libc.a` contains most other things. The library `libstdc++.a` contains many C++-specific functions.

[6] "`ar`" is short for "archive".

Program 11.7: oz.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }
  return ( rand()/(1.0+RAND_MAX) );
}

double normal () {
  int nroll = 12;
  double sum = 0;
  int i;
  for ( i=0; i<nroll; i++ ) {
    sum += rand01();
  }
  return ( sum - 6.0 );
}

void xydump( int npoints, double x[], double y[], char filename[] ) {
  FILE *output;
  int i;
  output = fopen( filename, "w" );
  for ( i=0; i<npoints; i++ ) {
    fprintf( output, "%lf %lf\n", x[i], y[i] );
  }
  fclose ( output );
}

double mean ( int nelements, double array[] ) {
  int i;
  double sum=0;
  for ( i=0; i<nelements; i++ ) {
    sum += array[i];
  }
  return ( sum/(double)nelements );
}

double stddev ( int nelements, double array[] ) {
  int i;
  double sum=0;
  double average;
  average = mean( nelements, array );
  for ( i=0; i<nelements; i++ ) {
    sum += pow(array[i]-average, 2);
  }
  return ( sqrt( sum/(nelements-1) ) );
}

int maxelement ( int nelements, double array[] ) {
  double max=0;
  int i, imax;
  for ( i=0; i<nelements; i++ ) {
    if ( array[i] > max ) {
      max = array[i];
      imax = i;
    }
  }
  return ( imax );
}

int minelement ( int nelements, double array[] ) {
  double min = 1.0e+30;
  int i;
  int imin;
  for ( i=0; i<nelements; i++ ) {
    if ( array[i] < min ) {
      min = array[i];
      imin = i;
    }
  }
  return ( imin );
}

void resethist (int nbins, int bin[]) {
  int i;
  for ( i=0; i<nbins; i++ ) {
    bin[i] = 0; // Reset all bins to zero.
  }
}

void addtohist ( int nbins, int bin[], double xmin, double xmax, double value ) {
  int binno;
  double binwidth;
  binwidth = (xmax-xmin)/(double)nbins;
  binno = (value-xmin)/binwidth;
  if ( binno >= 0 && binno < nbins ) {
    bin[binno]++; // Increment the appropriate bin.
  }
}

void histdump ( int nbins, int bin[], double xmin, double xmax, char * filename ) {
  FILE *output;
  int i;
  double binwidth;
  binwidth = (xmax-xmin)/(double)nbins;
  output = fopen( filename, "w" );
  for ( i=0; i<nbins; i++ ) {
    fprintf ( output, "%lf %d\n", xmin+binwidth*(double)i, bin[i] );
  }
  fclose ( output );
}
```

11.10. Using Your New Library

Now we have our new library, `liboz.a`, and we can use it when we compile programs. Say, for example, that we want to use one of our fancy new Oz functions in the `monkey.cpp` program. If `liboz.a` is in the current working directory, we might type:

```
g++ -Wall -o monkey monkey.cpp -L. -loz
```

The "`-L`" qualifier tells *g++* to look in an additional directory when trying to find libraries. (In this case, the directory is "`.`", which means the current working directory.) The "`-l`" qualifier says to link the program with the following library, where we leave off the "`lib`" prefix and the "`.a`" suffix on the library's name[7].

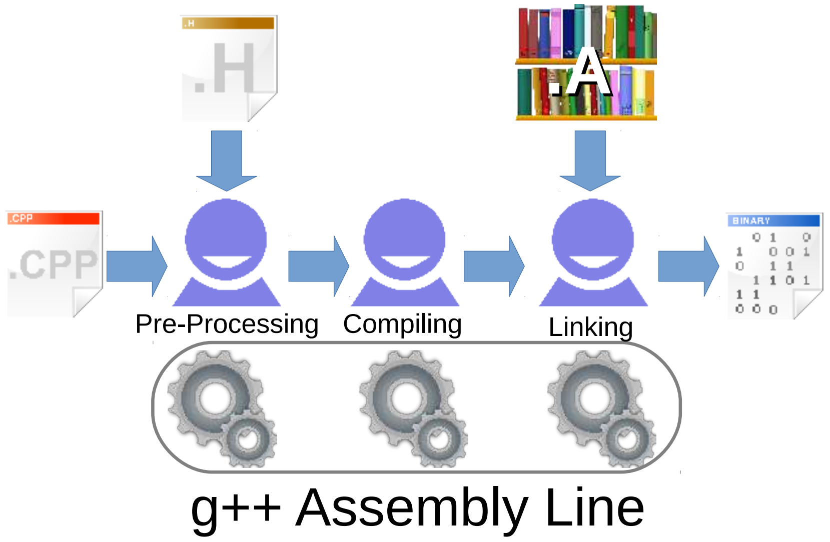


Figure 11.11: The *g++* assembly line processing `monkey.cpp`.

Exercise 57: Monkey Swarm

Create the files `oz.h` (Program 11.8), `oz.cpp` (Program 11.7), and `monkey.cpp` (Program 11.6).

Use `oz.cpp` to create a library named `liboz.a`. Compile the `monkey.cpp` program using this new library. Run the program. It should produce the file `monkey.dat` containing a histogram of the monkey positions after 1 hour of flying.

Plot the histogram using the *gnuplot* command `plot "monkey.dat" with boxes"`. The result should look like Figure 11.8.

[7] In the early days of the GNU project there was a library called "`libiberty.a`", so you could type "`-liberty`".

But what about...?

Can you look at a library and see what's inside it?

You can add more than one object file to a given library. The command "ar -t libsomething.a" will show you the names of the object files that were put into the library. Many of C's built-in functions live in a library called libc.a. The location of this file will vary from one type of computer to another, but if you can find it, try using "ar" to list the object files it contains. You'll see thousands of them.

To see the names of functions and symbols in the library's index, you can use the "nm" command. Each name will be shown with a one-letter symbol. The names of the functions in this library will be identified by a "T". The nm command is often useful when you're trying to figure out which library contains a particular function.

11.11. Function Prototypes

You might have noticed that we still haven't looked inside the header file "oz.h" that's used in monkey.cpp and oz.cpp. Here's what it looks like:

Program 11.8: oz.h

```
const int nmunchkin=1000; // Munchkin population.
const double dorothyspeed = 4; // Dorothy's walking speed, mph
const double poppytoxicity = 0.5; // Prob. of sleep after one minute's exposure.
const double poppyfieldsize = 1.0; // Width of poppy field, in miles.
const int nmonkeys = 1000; // Number of flying monkeys in swarm.
const double meanmonkeyspeed = 25; // mph, same as an unladen European swallow.
const double monkeyspeedspread = 5; // mph, std. dev. of monkey speeds.

double rand01 ();
double normal ();
void xydump( int npoints, double x[], double y[], char filename[] );
double mean ( int nelements, double array[] );
double stddev ( int nelements, double array[] );
int maxelement ( int nelements, double array[] );
int minelement ( int nelements, double array[] );
void resethist (int nbins, int bin[]);
void addtohist ( int nbins, int bin[], double xmin, double xmax, double value );
void histdump ( int nbins, int bin[], double xmin, double xmax, char filename[] );
```

It's probably not surprising that this file contains all of the constant definitions that we've been using, but what's the other stuff there for?

The second half of `oz.h` contains "function prototypes". These are one-line statements that define the syntax for using a function. They say what kind of value the function returns, how many arguments it wants, and what types of arguments.

By including `oz.h` at the top of our `monkey.cpp` file, we give *g*++ the information it needs to make sure we're using these functions correctly.

"Don't be afraid to make your own libraries!"

Source: *Wikimedia Commons*

Why is this necessary now that we're using a library? Until now, we've been defining our functions right at the top of our programs. The function definition itself tells *g*++ the function's syntax. Now that we've moved the function definitions into a library, we need to create these function prototypes to give *g*++ that information.

Function prototypes make up much of the header files we've been using. Files like `stdio.h` and `math.h` contain prototypes for functions like `printf` and `sqrt`.

11.12. Static versus Dynamic Libraries

There are actually two different kinds of libraries: static and dynamic libraries. Much of what we've said so far applies only to static libraries. A static library has a name like "libsomething.a", with ".a" standing for "archive". Static libraries are used by the linker as we described above.

Dynamic libraries are slightly different. When a program uses dynamic libraries, the binary code for the functions you use isn't physically inserted into the binary file created by the linker. Instead, a reference is inserted into the file. This reference says that, when the program is run, the function should be loaded as needed from a dynamic library. Dynamic library files usually have names ending in "`.so`", "`.dll`", or "`.dylib`", depending on what kind of computer you're using.

Why would you want to use dynamic libraries instead of static libraries? There are several reasons:

- Dynamic libraries save disk space. If every program contained its own copy of "`printf`" a lot of space would be wasted. With dynamic libraries, there's only one copy of these functions.

- Dynamic libraries make upgrades easy. Imagine that there’s a serious bug in an old version of a library, and you want to install a newer version. If it’s a static library, it’s not sufficient to just install the new “`libsomething.a`” file. Programs that were compiled with the old static library will still have buggy functions inside them. In order to give all of your programs the benefit of the new library, you’d need to recompile all of them, so that the new, un-buggy functions from the library would be copied into the new binary files.

  With Dynamic libraries all you need to do is install a new “`libsomething` (or “`.dll`” or “`.dylib`”) file. Any programs that use the library will automatically, immediately, see the benefit of the upgrade, without your needing to do anything else. This can be very important if the bug is a security hole.

- Dynamic libraries save memory. When you run a program, it gets copied into memory. Just as with disk space, multiple copies of library functions waste memory. When programs use dynamic libraries, the operating system is smart enough to load only one copy of each library into memory. This copy is shared by all of the programs that need that library.

Buddy Ebsen (right), later the star of *The Beverly Hilbillies*, originally had the role of the Tin Man in the 1939 movie version of *The Wizard of Oz*. He resigned because of health problems, possibly due to the aluminum dust that was part of his costume.

Source: Wikimedia Commons

For all of those reasons, most of the programs installed on your computer use at least some dynamic libraries.

OK, so dynamic libraries sound great. But what’s the down side? Well, here’s one: What happens if you copy your program to another computer that doesn’t have all of the dynamic libraries that the program needs?

When compiling a program with dynamic libraries, it’s also possible to specify a particular version of a library, or even to say where we’re going to expect to find the library on disk. These things can also make a binary file un-portable if another computer has a different version of a library, or if the library is stored in a different location on disk.

The procedures for creating and using shared libraries will vary significantly from one kind of computer to another, so we unfortunately won’t be able to cover them here.

11.13. Conclusion

Whew! That was quite an adventure, but Dorothy has finally clicked her ruby slippers[8] together three times, and returned safely home to Kansas. Even better, we now know how to create our own libraries.

Creating libraries of functions can make it easier for you to re-use functions you've written. As you go further in programming, you'll also discover many useful libraries that have been written by other programmers. For example:

- The GNU Scientific Library is a rich collection of functions relevant to math, science, and engineering.
- LAPACK (Linear Algebra PACKage) is standard library for dealing with problems in "linear algebra" (matrices and such).
- FFTW ("The Fastest Fourier Transform in the West") is the go-to library for Fourier transforms.
- libjpeg is a free library for reading and writing jpeg files.

The details of installing and using these will depend on the kind of computer you're using, and which C compiler you use.

Figure 11.12: Dorothy's ruby slippers, in the Smithsonian National Museum of American History.

Source: *Wikimedia Commons*

[8] *Silver* slippers in the original book.

Practice Problems

1. Create a header file named `conversions.h` that contains three functions named `to_celsius`, `to_meters`, and `to_kilograms` that each take one `double` argument and return a `double` value. These functions should (respectively) convert a given temperature from Fahrenheit to Celsius; convert a given length from Feet to Meters; and convert a given weight in Pounds into a mass in Kilograms. Use the conversion formulas shown in Figure 11.13

$$Celsius = \frac{5}{9}(Fahrenheit - 32)$$

$$Meters = \frac{Feet}{3.281}$$

$$Kilograms = \frac{Pounds}{2.205}$$

Figure 11.13: Conversion formulas.

   Now write a program named `conversions.cpp` that uses your newly-written header file. The program should present the user with a menu like this:

```
Choose a conversion:
1 for Fahrenheit to Celsius
2 for Feet to Meters
3 for Pounds to Kilograms
```

   and then ask the user for the value she or he wants to convert. The program should print the converted value in a friendly form, like "`100.000000 Fahrenheit is 37.777778 Celsius`". Be sure you give users a friendly error message if they enter a number that's not on the menu.

2. An important principle in optics is Snell's Law, illustrated in Figure 11.14. Snell's Law tells us how much a ray of light will bend when it passes from one material to another (from air to glass, for example). In the terms used in the figure, Snell's Law says that:

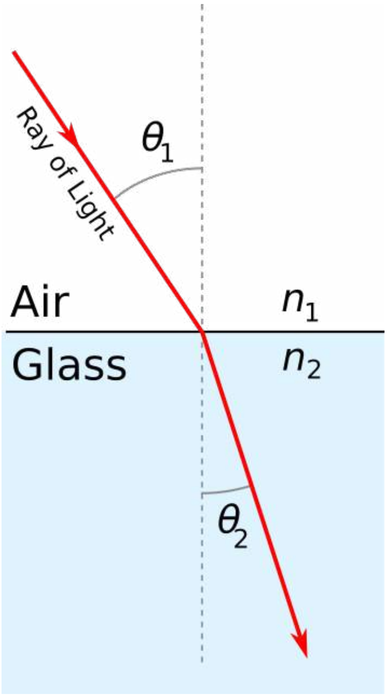


Figure 11.14: Snell's law tells us how light bends when going from one medium to another.

Adapted from: Wikimedia Commons

$$sin(\theta_2) = \frac{n_1}{n_2} sin(\theta_1) \qquad (11.1)$$

   where n_1 and n_2 are properties of the two materials, called their "indices of refraction". If we know the index of refraction of each material and we know the angle of the incoming ray, we can predict the angle it will have after entering the second material.

   The index of refraction will often depend on the light's color. For borosilicate glass, we can estimate the index of refraction using this rule of thumb:

$$n \simeq 1.5046 + \frac{0.00420}{wavelength^2} \qquad (11.2)$$

   where the light's color is specified by its wavelength measured in microns[9].

   Write a header file named `optics.h` that contains three functions related to Snell's Law. The first function should use Equation 11.2 above to calculate the index of refraction in borosilicate glass for light of a given wavelength. It should start out like this:

[9] See "Cauchy's Equation": Wikipedia.

```
double refractive_index ( double wavelength ) {
```

The second function should start out like this:

```
double refraction_angle ( double n1, double n2, double theta1 ) {
```

It takes the values of n_1, n_2, and θ_1 and returns the value of θ_2, using Equation 11.1. That equation will give you the value of $sin(\theta_2)$. Once you have that, you can use C's `asin` ("arcsine") function to find the value of θ_2 itself. This is the value that the `refraction_angle` function should return.

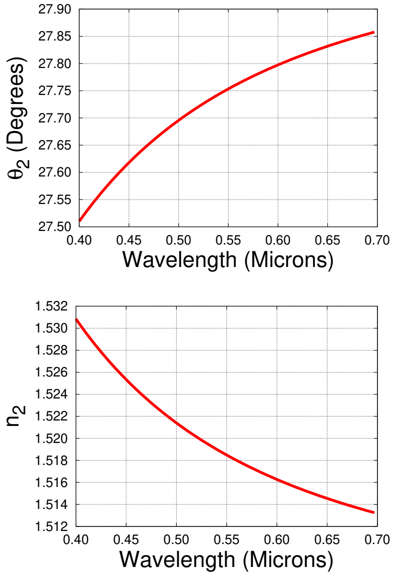


Figure 11.15: Graphs produced by "`plot "optics.dat" with lines`" (top) and "`plot "optics.dat" using 1:3 with lines`" (bottom).

The third function should convert angle measurements from radians to degrees. (Remember that C's trigonometry functions measure angles in radians.) To convert an angle from radians to degrees, just multiply by 360 and divide by 2π. The function should start out like this:

```
double to_degrees ( double radians ) {
```

After you've written your `optics.h` file, make a program named `optics-test.cpp` that looks like this:

```
#include <stdio.h>
#include <math.h>
#include "optics.h"
int main () {
  int i;
  double wmin = 0.4; // microns
  double wmax = 0.7; // microns
  double wavelength, wstep;
  double n1 = 1, n2;
  double theta1 = M_PI/4, theta2; // theta1 is 45 degrees
  int nsteps = 100;

  wavelength = wmin;
  wstep = (wmax-wmin)/nsteps;
  for ( i=0; i<nsteps; i++ ) {
    n2 = refractive_index( wavelength );
    theta2 = refraction_angle( n1, n2, theta1 );
    printf( "%lf %lf %lf\n", wavelength, to_degrees( theta2 ), n2 );
    wavelength += wstep;
  }
}
```

This program will try out the functions you've written by stepping through 100 different color values[10] and finding the outgoing angle (θ_2) for each of them, expressed in degrees. If you run the program like this:

```
./optics-test > optics.dat
```

you can plot the program's output and you should see graphs like those shown in Figure 11.15.

3. This problem uses the `oz.h` header file and `liboz.a` library that we created in Section 11.9. Make sure you've created these before you start!

Is your distribution *normal?*
Source: Wikimedia Commons

In Program 11.6 we introduced the histogram functions `addtohist`, `resethist`, and `histdump`. In Program 11.1 we introduced the function `normal`, which generates pseudo-random number distributed "normally". These functions were then included in `liboz.a`. In order to use the histogram functions in a program, you need to first define a variable that will be the histogram's bins. You might do something like this:

```
const int nbins=50;
int bin[nbins];
```

The lines above would define a 50-element array named `bins`.

Write a program named `testnormal.cpp` that uses these functions. The program will need a line like:

```
#include "oz.h"
```

to get the necessary header file. Have the program define a 50-element array like `bin` above, and have it use `resethist` to reset all of the values in the array to zero.

Then have the program generate 100,000 pseudo-random numbers, using the `normal` function. Each time a number is generated, add it to the histogram using the `addtohist` function. Tell `addtohist` that the minimum and maximum values to be histogrammed are -3 and 3, respectively.

After generating all of the numbers and adding them to the histogram, dump the histogram's contents into a file using the `histdump` function. Call the output file `testnormal.dat`.

[10] 0.4 to 0.7 microns is the range of wavelengths in visible light.

Compile your program, linking it against the `liboz.a` library so that it can find the necessary functions.

After running your program, use *gnuplot* like this to see the histogram:

```
plot "testnormal.dat"
```

Does it look like a "normal" distribution?

4. This problem uses the `oz.h` header file and `liboz.a` library that we created in Section 11.9. Make sure you've created these before you start! We're going to be adding some functions to the `liboz.a` library.

   In Chapter 9 we used two functions named `to_radians`, to convert degrees into radians, and `time_of_flight`, to find the time of flight of a projectile fired with a given angle and speed.

   Add these functions to `oz.cpp`, and add prototypes for them to `oz.h`. You'll also need to add the following to `oz.cpp`:

```
const double g = 9.81; // Acceleration of gravity.
```

   Re-build the `liboz.a` library.

   Test your newly-rebuilt library by compiling the following program named `oztest.cpp`:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#include "oz.h"

int main () {

    double angle = 45.0; //degrees.
    double v0 = 27.0; // m/s.

    printf ( "%lf\n", time_of_flight( v0, to_radians(angle) ) );
}
```

   The result should match what we saw when using the same angle

Look out below!

Source: Wikimedia Commons

and speed in Chapter 9: about 3.9 seconds.

```
8.000000 4.000000 1.000000
5.000000 7.000000 5.000000
1.000000 0.000000 3.000000
Trace is 18.000000
Determinant is 121.000000
```

Figure 11.16: Output from the `mat3d-test.cpp` program.

5. Create a library for doing calculations with 3 × 3 matrices. Your library should contain three functions with prototypes that look like this:

```
double trace ( double m[3][3] );
double determinant ( double m[3][3] );
void printmat ( double m[3][3] );
```

The `trace` function should return the trace of the matrix `m`, defined as in Problem 4 in Chapter 6. The `determinant` function should return the matrix's determinant, as defined in the same problem. The `printmat` function should neatly print out the matrix's elements as a 3×3 grid of numbers.

Put the functions into a file named `mat3d.cpp` and put the prototypes into a file named `mat3d.h`. Then build your library using the `g++ -c` and `ar -csr` commands described in Section 11.9 earlier in this chapter. Call your new library `libmat3d.a`.

Then use the following program to test your newly-created library.

```
#include <stdio.h>
#include "mat3d.h"
int main () {
  double matrix[3][3] = {{8,4,1},{5,7,5},{1,0,3}};

  printmat( matrix );
  printf ( "Trace is %lf\n", trace( matrix ) );
  printf ( "Determinant is %lf\n", determinant( matrix ) );
}
```

Just copy this program into a file named `mat3d-test.cpp` and compile it with your library as described in Section 11.10 above.

When you run the `mat3d-test` program its output should look exactly like Figure 11.16.

6. Create a file named `resistor.cpp` that contains the `current` and `power` functions from Section 9.4 in Chapter 9. Then create a new function named `voltage` that calculates the voltage across a resistor when given the resistance and the current, using the formula *voltage* = *current* × *resistance*, and add this function to your `resistor.cpp` file. (Make the current the first argument you give the function.)

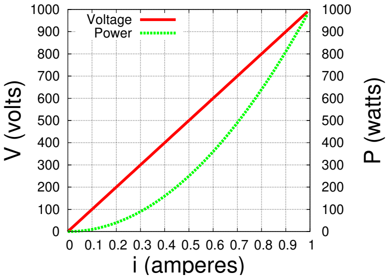


Figure 11.17: Graph of the output from `resistor-check.cpp`.

Create another file named `resistor.h` and put prototypes for the three functions into that file. Use the `g++ -c` and `ar -csr` commands (described in Section 11.9 earlier in this chapter) to turn your `resistor.cpp` file into a library named `libresistor.a`.

Check that your new library works by compiling the following test program, called `resistor-check.cpp`:

```
#include <stdio.h>
#include "resistor.h"
int main () {
  double r = 1000; // ohms.
  double imin = 0; // amperes.
  double imax = 1; // amperes.
  double v, p, i, istep;
  int n;

  i = imin;
  istep = (imax - imin)/100.0;
  for ( n=0; n<100; n++ ) {
    v = voltage ( i, r );
    p = power ( v, r );
    printf ( "%lf %lf %lf\n", i, v, p );
    i += istep;
  }
}
```

Compile this program, using your newly-created `libresistor.a` library, and run it like this:

```
./resistor-check > resistor-check.dat
```

Then use *gnuplot* to plot the output from your program like this:

```
plot "resistor-check.dat" using 1:2 with lines, "" using 1:3 with lines
```

This should produce a graph similar to Figure 11.17.

12. *Structures*

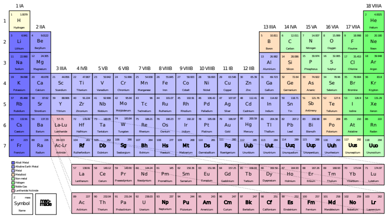
12.1. Introduction

Before the widespread use of fingerprinting, investigators in Europe and the U.S. used a system called the *portrait parlé* to identify criminals. Introduced in the 1880s by Alphonse Bertillon, this was a detailed written description of a person's physical characteristics. You'll find many references to the *portrait parlé* in Victorian detective fiction by writers like Gaston Leroux.

A "Bertillon card" describing the physical characteristics of Alphonse Bertillon himself.

Source: *Wikimedia Commons*

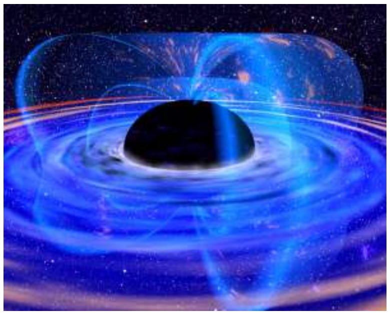
Anything studied by researchers will probably have more than one interesting property. In Chapter 6 we saw that researchers often make several measurements at the same time. For example, a census-taker visiting a house might record the number of children, the household income, the size of the house, and so forth. We could store each of these quantities in a separate variable, but we know that they're all actually properties of a single household. It might be convenient if that reality could be reflected in our programs.



Each chemical element has many properties: Atomic number, atomic mass, ionization energies, and so forth.

Source: *Wikimedia Commons*

Also, in Chapter 9 we learned that C functions can only return a single value. It would sometimes be useful to return several related values at once. Imagine, for example, that we were working with 3-dimensional vectors, which have x, y, and z components. Wouldn't it be great if we had a function that could add two vectors and return all three components of their sum?



Even black holes, perhaps the most featureless bodies in the universe, have at least three properties: mass, angular momentum, and charge.

Source: *Wikimedia Commons*

Fortunately, C allows us to do both of these things through a mechanism called "structures". Structures let us store several related measurements in one convenient place. Let's look at how to create and use structures by working through some examples.

12.2. The “struct” Statement

If we were comparing Virginia with other states, we might write a program that had variables like this:

```
int va_population;
double va_area;
double va_income;
```

It’s tedious to create all of these variables, and could rapidly become confusing as we added more properties of the state, or more states. C provides us with a better way to do it, by allowing us to create custom-made variables that pack several properties together in one place. This is done with the “struct” statement:

```
struct {
  int population;
  double area;
  double income;
} va;
```

The statement above defines a new variable, `va`, that has, packed within it, several values of different types. The variable `va` isn’t an `int` or a `double` or any of the other types we’ve used so far. It’s a “structure”. This structure is a completely new, custom-made type that contains whatever we want it to contain.

Once we’ve defined our `va` variable, we can set its properties like this:

```
va.population = 8326289;
va.area = 42774.2;
va.income = 61044;
```

The dot operator (“`.`”) singles out one of the properties of the structure. Similarly, we can use the properties of the structure in just the same way we’ve used variables in the past:

```
if ( va.population < 8491079 ) {
  printf ("Virginia has fewer people than New York City\n");
}
```

We could even define an array of such structures, just as we’ve defined arrays of `int` or `double` variables:

```
struct {
  char name[20];
  int population;
  double area;
  double income;
  double birthrate;
  double deathrate;
} state[50];
```

Here we've defined an array of 50 structures, one for each of the 50 states. We've also added some more properties, such as the state's name. We can set the properties of an individual state by referring to it by its element number:

```
snprintf( state[0].name, 20, "Virginia" );
state[0].population = 8326289;
state[0].area = 42774.2;
state[0].income = 61044;
...etc.
```

This would make it easy to loop through all of the states:

```
for ( i=0; i<50; i++ ) {
  printf ( "Pop. of %s is %d\n", state[i].name, state[i].population );
}
```

What if we wanted to use the same structure for other variables? Say, for example, we wanted to store census data for a group of 100 countries. We could just re-type the structure definition:

```
struct {
  char name[20];
  int population;
  double area;
  double income;
  double birthrate;
  double deathrate;
} state[50];

struct {
  char name[20];
  int population;
  double area;
  double income;
  double birthrate;
  double deathrate;
} country[100];
```

There's a better way to do it, though. Instead of re-typing the "struct" statement, we can use "typedef" to define a name for this kind of structure.

12.3. Using "typedef"

If we want to re-use our structure, we can do something like this:

```
typedef struct {
  char name[20];
  int population;
  double area;
  double income;
  double birthrate;
  double deathrate;
} census;

census state[50];
census country[100];
```

The statements above define a new variable type named "census". We can use this new type to define variables, just like the int and double types we've used before. In the example above, we define two census arrays, one of 50 states and one of 100 countries.

typedef and struct are so often used together that many textbooks lump them into a single statement, "typedef struct", but they can be used separately too.

For example, typedef can be used to define an alternative name for any variable type. For example:

```
//Define aliases for some types:
typedef int funds;
typedef double weight;
typedef int days;

//Use these aliases to define some variables:
funds bank_balance;
weight fish_per_month[12];
days til_christmas;
```

This may make it easier for you to re-define your variables later on. Say, for example, that you've made so much money that you now need to use a “`long int`”[1] to count your fortune! If your program uses the “funds” type for all of your accounting variables, then you’ll only need to change one line: the `typedef` statement that defines “funds”.

[1] In C, “`long int`” is a variable type that can hold larger number than a plain,old `int` is capable of holding.

12.4. Using Vectors in Programs

Portrait of Rene Descartes, French philosopher, mathematician, and scientist, for whom the “cartesian” coordinate system is named. As a philosopher, he’s famous for his statement “*cogito ergo sum*” (“I think, therefore I am”).

Source: Wikimedia Commons

A “vector” is something that has both a magnitude and a direction. Physical properties like velocity and acceleration are vectors. Even though an eastbound car and a westbound car may have the same speed, their velocities are different, since they’re going in different directions.

In many fields of science and mathematics, it’s very useful to be able to define and use vectors in computer programs. Vectors are often represented by their x, y, and z components in a Cartesian coordinate system.

We could use our new-found knowledge of the `struct` and `typedef` commands to make it easy to write programs that deal with vectors. All we need to do is define a new variable type that’s designed to hold a vector’s three cartesian components.

```
typedef struct {
  double x;
  double y;
  double z;
} Vector;
```

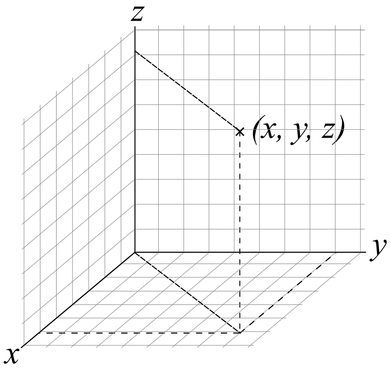


Figure 12.1: The cartesian coordinate system.

Source: Wikimedia Commons

We can then define variables that have the new type `Vector`. For example, we might define a vector named “`velocity`”, and give it a magnitude of 60 mph along the x axis:

```
Vector velocity;

velocity.x = 60;
velocity.y = 0;
velocity.z = 0;
```

When initializing a “structure” variable, we can also take advantage of the following shortcut:

```
Vector velocity = {60,0,0};
```

The items listed in the curly brackets are the initial values for each of the structure's properties, in the same order they appear in the `struct` statement.

Now that we can define our own variable types, we can write functions that return more information. In C, functions can only return one "value", but that value can be a structure. Program 12.1 shows a few examples of this, with our new variable type highlighted. Notice that we can use our new type for arguments to our functions or the values the functions return.

The `add_vector` function defined in Program 12.1 adds two vectors together and returns their sum as a third vector. The `scale_vector` function "scales" a vector by multiplying its magnitude by some amount but leaving its direction unchanged. The `print_vector` function just prints out a vector's components.

We could generalize our vector functions to any number of dimensions, and reduce the amount of repetitive typing in them, by replacing the x, y, and z components with a three-element array of components:

```
const int dimension = 3;

typedef struct {
  double x[dimension];
} Vector;
```

Instead of x, y, and z, we would then use `x[0]`, `x[1]`, and `x[2]`. A function like `scale_vector` would then look like this:

```
Vector scale_vector ( double r, Vector v ) {
  Vector vscale;
  int i;
  for ( i=0; i<dimension; i++ ) {
          vscale.x[i] = r * v.x[i];
  }
  return ( vscale );
}
```

This would also require that we change the way we initialize our vectors:

```
  Vector v1 = {{0,0,1}};
  Vector v2 = {{1,0,0}};
```

(Notice the double curly brackets.) This says that the first (and only, in this case) property of v1 is an array containing the values 0, 0, and 1. Compare this to our earlier case with x, y, and z properties.

Program 12.1: vector.cpp

```
#include <stdio.h>
#include <math.h>

typedef struct {
  double x;
  double y;
  double z;
} Vector;

Vector scale_vector ( double r, Vector v ) {
  Vector vscale;
  vscale.x = r * v.x;
  vscale.y = r * v.y;
  vscale.z = r * v.z;
  return ( vscale );
}

Vector add_vectors ( Vector v1, Vector v2 ) {
  Vector sum;
  sum.x = v1.x + v2.x;
  sum.y = v1.y + v2.y;
  sum.z = v1.z + v2.z;
  return ( sum );
}

void print_vector ( Vector v ) {
  printf ( "x = %lf\n", v.x );
  printf ( "y = %lf\n", v.y );
  printf ( "z = %lf\n", v.z );
}

int main () {

  Vector v1 = {0,0,1};
  Vector v2 = {1,0,0};

  printf ( "Vector 1:\n" );
  print_vector( v1 );

  printf ( "Vector 2:\n" );
  print_vector( v2 );

  printf( "Sum of vectors:\n" );
  print_vector( add_vectors( v1, v2 ) );

  printf ( "Pi times Vector 1:\n" );
  print_vector( scale_vector( M_PI, v1 ) );
}
```

Exercise 58: What's Your Vector Victor?

Starting with `vector.cpp`, modify the program so that it prompts the user for the x, y, z components of two vectors, then prints out the components of the sum of the two vectors.

But what about...?

When we studied arrays in Chapter 6 and character strings in Chapter 8 we found that we couldn't just make two arrays equal by saying "array1 = array2", or compare arrays the way we'd compare single variables. Are there similar restrictions on structures?

First, regarding setting structures equal, there's good news. The following will work fine:

```
typedef struct {
    double width;
    double height;
} Rectangle;

Rectangle r1, r2;

r1.width = 8.5;
r1.height = 11.0;

// Make r2 equal r1:
r2 = r1;
```

Another easy shortcut for setting the value of a structure is this:

```
// Set the width and height of rectangle r1:
r1 = (Rectangle){8.5,11.0};
```

Regarding the comparison of two structures, the answer is the same as for arrays. The usual comparison operators (==, <, >, *et cetera*) won't work. You'll need to compare the properties of your structures yourself. For example, with our rectangles above we might write:

```
if ( r1.width == r2.width && r1.height == r2.height ) {
   printf ( "They're equal!\n" );
}
```

12.5. Gravitation

Okay, now we have vectors so let's do something with them. Imagine you have two masses, say the Earth and the Sun. Newton's Law of Gravitation tells us that each of these bodies will attract the other with a force whose magnitude is given by:

$$F = G\frac{m_{earth}m_{sun}}{r^2} \tag{12.1}$$

where m_{earth} and m_{sun} are the masses of the bodies, and r is the distance between them. G is Newton's gravitational constant, equal to about $6.67 \times 10^{-11} \frac{m^3}{kgs^2}$. Earth and Sun pull on each other with equal force, but in opposite directions.

Isaac Newton's grave in Westminster Abbey.

Source: Wikimedia Commons

In principle, if we know a body's initial position and velocity, its mass, and the forces acting on it, we can predict its future motion. Could we simulate the motion of the Earth and the Sun? Let's try! In the following, we won't spend much time discussing the physics of the problem. We'll focus on the programming challenges it presents.

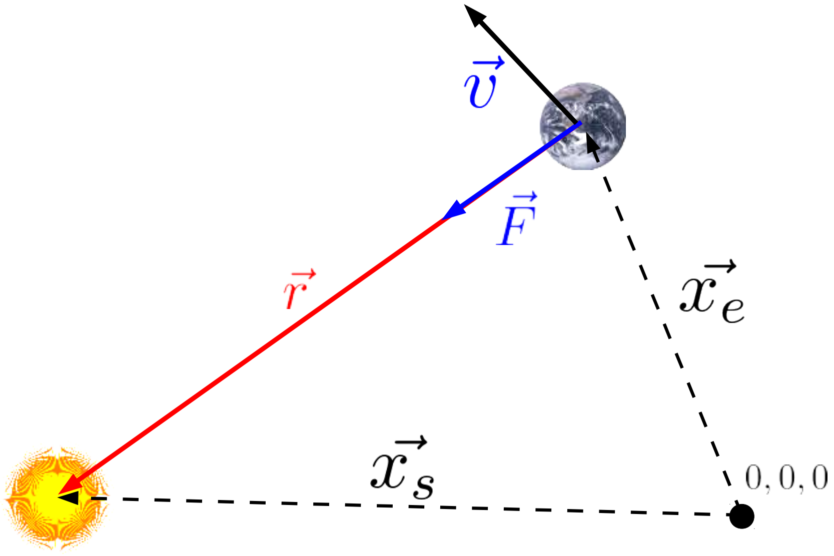


Figure 12.2: The earth-sun system, showing some of the vector quantities we might want to use in our program. $\vec{F}$ represents the force of the sun on the earth. The origin of our cartesian coordinate system is shown in the lower right corner. The vectors x_e and x_s represent the position of the earth and sun in this coordinate system.

First, we'll need a little more information from Newton. He also tells us that force and acceleration are related in this way:

$$\vec{F} = m\vec{a} \tag{12.2}$$

where $\vec{F}$ is a vector representing the magnitude and direction of the total force on an object, m is the object's mass, and $\vec{a}$ is the object's acceleration. In order to predict the motion of an object, we'll need to know its acceleration. We can get this by rearranging Equation 12.2:

$$\vec{a} = \frac{\vec{F}}{m} \tag{12.3}$$

Next, we'll need to know how acceleration affects an object's motion. Acceleration is the rate of change of velocity, so we might guess that after a small amount of time, Δt, the object's velocity will change by $\vec{a}\Delta t$. We also know that velocity is the rate of change of position, so we might approximate the change of position during a short time as $\vec{v}\Delta t$.

Given this chain of relationships, we can start with an object's initial position and velocity, then move forward in time by small steps and follow the changes in the object's position.

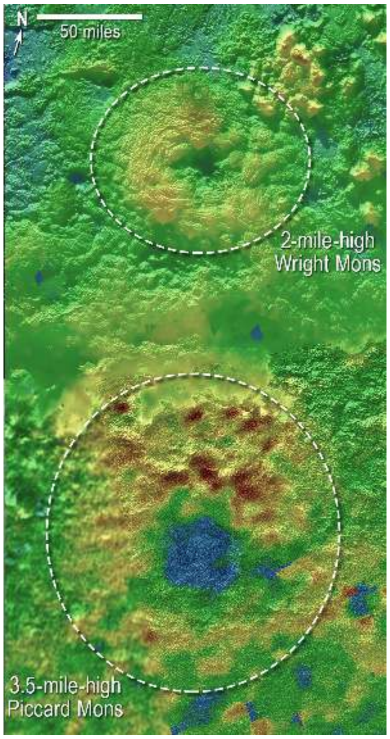
Both the Earth and the Sun move in response to their mutual gravitational attraction. We'll be tracking several properties of each of these objects: mass, velocity, position, and the force acting on the object. This sounds like a good place to use C's structures.

Here are the structures we'll be using, with convenient "typedef" names given to them:

```
const int dimension = 3;

typedef struct {
  double x[dimension];
} Vector;

typedef struct {
  double mass;
  Vector x; // Position
  Vector v; // Velocity
  Vector f; // Force on the body
} Body;
```

As before, we've defined a new type of variable called Vector to hold vectors. Now we add a variable called Body that holds information about one of the bodies we'll be tracking. Notice that the Body type has several properties that are of the Vector type.

In addition to the vector functions we've already written, we'll need a



The photo above shows a false-color image of two icy "cryovolcanos" on Pluto. After nearly ten years in space, NASA's *New Horizons* space probe whizzed past Pluto during a few hours in 2015, snapping thousands of photos and taking measurements of many kinds. This kind of precise navigation demands highly sophisticated computer simulations.

Source: NASA/JHUAPL

few others:

```
Vector invert_vector ( Vector v ) {
  Vector inverse;
  int i;
  for ( i=0; i<dimension; i++ ) {
    inverse.x[i] = -v.x[i];
  }
  return ( inverse );
}

Vector subtract_vectors ( Vector v1, Vector v2 ) {
  return ( add_vectors( v1, invert_vector(v2) ) );
}

double vector_magnitude ( Vector v ) {
  double size2=0;
  int i;
  for ( i=0; i<dimension; i++ ) {
    size2 += v.x[i]*v.x[i];
  }
  return ( sqrt( size2 ) );
};
```

The function `subtract_vectors` is the companion to the `add_vectors` function we saw earlier. Subtraction is just equivalent to adding an inverse, so we implement the subtraction function by defining a new `invert_vector` function and then using it along with `add_vectors` to do the subtraction. Finally, we'll need to know the size of vector quantities, so we define a new function named `vector_magnitude` to do this.

Armed with all of these new tools, we're now ready to tackle the weighty problem of swinging the Earth around the Sun. The result is Program 12.2. Here we've swept all of the functions we've discussed so far into a header file named `gravity.h`.

The program steps through time in 10,000-second jumps (about 3 hours). In each "time slice" the program updates the position and velocity of Earth and Sun, based on the force of their mutual gravitational attraction, then writes out the current time and the position of each body.

Program 12.2: gravity.cpp

```
#include <stdio.h>
#include <math.h>

#include "gravity.h"

int main () {
  // All units in kilograms, meters, kilograms, and seconds.
  //            mass     position         velocity        force
  Body sun   = {2.0e+30, {{0,0,0}},       {{0,0,0}},      {{0,0,0}}};
  Body earth = {1.5e+24, {{1.5e+11,0,0}}, {{0,0,3.2e+4}}, {{0,0,0}}};

  double distance, force, deltat=1e+4; // About 3 hours.
  int i, nsteps = 10000;
  double G = 6.67e-11;
  Vector r, deltax, deltav;

  for ( i=0; i<nsteps; i++ ) {
    // Find forces from law of gravitation:
    r = subtract_vectors( earth.x, sun.x );
    distance = vector_magnitude( r );
    force = G*sun.mass*earth.mass/(distance*distance);
    sun.f = scale_vector ( force/distance, r );
    earth.f = invert_vector( sun.f );

    // Update positions and velocities for next step:

    // Sun
    deltax = scale_vector ( deltat, sun.v );
    sun.x = add_vectors( sun.x, deltax );
    deltav = scale_vector ( deltat/sun.mass, sun.f );
    sun.v = add_vectors( sun.v, deltav );

    // Earth
    deltax = scale_vector ( deltat, earth.v );
    earth.x = add_vectors( earth.x, deltax );
    deltav = scale_vector ( deltat/earth.mass, earth.f );
    earth.v = add_vectors( earth.v, deltav );

    // Write out current time and positions
    printf ("%lf ", deltat*(double)i );
    print_vector ( sun.x );
    print_vector ( earth.x );
    printf ("\n");
  }
}
```

If we save the program's output into a file we can plot the results with *gnuplot*. Some of the results are shown in Figures 12.3 and 12.4. As you can see, our simulation gives earth a rather rough ride. The orbit is approximately regular, but doesn't come back quite to the place it started. If we asked the program to simulate more time steps the orbit would eventually become a spiral.

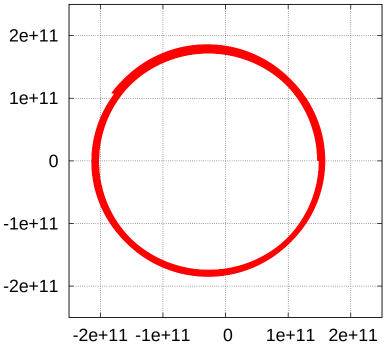


Figure 12.3: The Earth's orbit as approximated by Program 12.2.

By looking at the period of Earth's movement along the X-axis of our coordinate system, we can estimate the length of Earth's year. An actual year is about 3×10^7 seconds long, but our simulated Earth has a rather longer year of about 4.5×10^7 seconds.

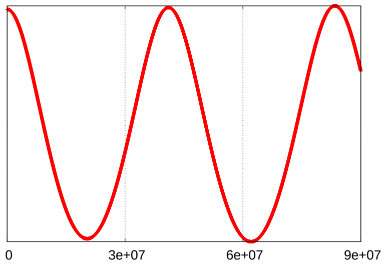


Figure 12.4: The Earth's position on the X-axis as a function of time, as approximated by Program 12.2.

Don't use this program to navigate your space probe to Pluto! We could probably improve the program in several ways. We could use more precise values for the mass, distance, and initial velocity of the Sun and Earth, for example. Still, for a relatively simple program the results aren't too bad.

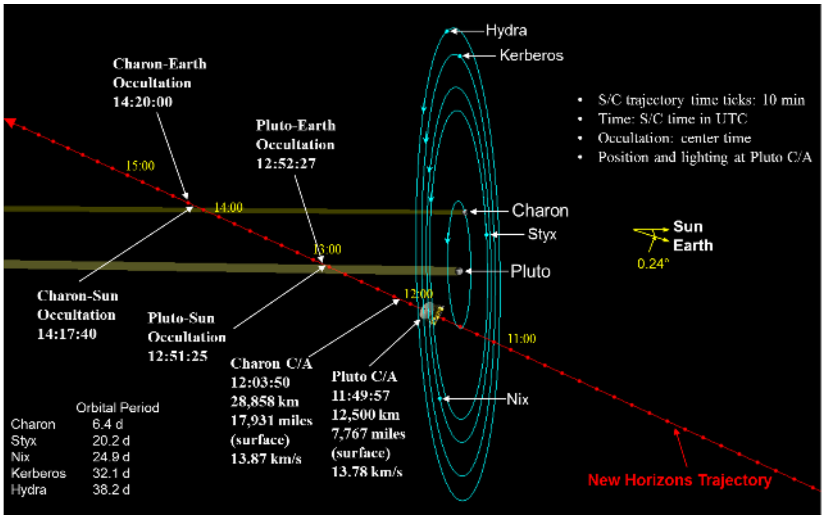

Hydra
Kerberos
Charon-Earth Occultation 14:20:00
Pluto-Earth Occultation 12:52:27
S/C trajectory time ticks: 10 min
Time: S/C time in UTC
Occultation: center time
Position and lighting at Pluto C/A
15:00
14:00
13:00
12:00
11:00
Charon
Styx
Pluto
Sun
Earth
0.24°
Charon-Sun Occultation 14:17:40
Pluto-Sun Occultation 12:51:25
Charon C/A 12:03:50 28,858 km 17,931 miles (surface) 13.87 km/s
Pluto C/A 11:49:57 12,500 km 7,767 miles (surface) 13.78 km/s
Nix
Orbital Period
Charon 6.4 d
Styx 20.2 d
Nix 24.9 d
Kerberos 32.1 d
Hydra 38.2 d
New Horizons Trajectory

Figure 12.5: New Horizons' trajectory through the Pluto system is a stunning example of precise navigation.

Source: NASA/JHUAPL

But what about...?

Yow! that's all very impressive, but those vector function things are really hard to read:

```
sun.x = add_vectors( sun.x, deltax );
```

Wouldn't it be so much nicer if you could just write "`sun.x = sun.x + deltax`"? If you use the extra features of C++, you can!

C++ has extra capabilities beyond those of plain C. One of them is called "operator overloading". This allows you to define how operators like + and − act when used with structures.

All we need to do is add the following to the `gravity.h` file used by Program 12.2:

```
Vector operator+( Vector v1, Vector v2 ) {
  return ( add_vectors ( v1, v2 ) );
}

Vector operator-( Vector v1, Vector v2 ) {
  return ( subtract_vectors ( v1, v2 ) );
}
```

This tells the compiler that, when it sees you adding two vectors in an expression like "`vsum = v1 + v2`", it should pass `v1` and `v2` to your `add_vectors` function and use that to add the vectors. The "`operator-`" statement does the analogous thing for subtraction.

12.6. Complex Numbers

Like vectors, another natural use for structures is the representation of complex numbers. These are numbers of the form $a + ib$, where a and b are real numbers, and i is $\sqrt{-1}$. The value of a is called the "real part" of the complex number, and b is its "imaginary part".

Using `struct` and `typedef` we can define a new variable type for holding complex numbers:

```
typedef struct {
  double re;  // real part
  double im;  // imaginary part
} Complex;
```

The "magnitude" of a complex number represents its size, taking both of its components (real and imaginary) into account. The magnitude is just the same as though real and imaginary components were the x and y components of a cartesian vector: magnitude $= \sqrt{\mathrm{Re}^2 + \mathrm{Im}^2}$. Similarly, complex numbers add in the same way that 2-dimensional vectors add.

Strangeness enters the picture when we begin multiplying complex numbers. Since they involve multiples of i, and $i \times i = -1$, minus signs begin to appear in surprising places.

We could use our new variable type to define a few functions for operating with complex numbers:

```
double magnitude_complex( Complex z ) {
  return sqrt( z.re*z.re + z.im*z.im );
}

Complex multiply_complex ( Complex a, Complex b ) {
  Complex result;
  result.re = a.re*b.re - a.im*b.im;
  result.im = a.im*b.re + a.re*b.im;
  return ( result );
}

Complex add_complex ( Complex a, Complex b ) {
  Complex result;
  result.re = a.re + b.re;
  result.im = a.im + b.im;
  return ( result );
}
```

Mathematician Robert W. Brooks, who along with Peter Matelski discovered the Mandelbrot set in 1978. The set was later named in honor of Benoit Mandelbrot, who studied it extensively.

Source: Wikimedia Commons

12.7. The Mandelbrot Set

Mathematicians love to play games with numbers. Let's try one here. Here are the rules:

Take two numbers, c and z_0. Pick any number you want for c, but set z_0 equal to zero. Now write down the value of $z_0^2 + c$. Call this new number z_1. Now write down the value of $z_1^2 + c$. Call this z_2. Keep doing this for more and more z_n values. We might expect that each z would be bigger than the last.

We coult write a little program to test this. Let's pick $c = 1$ as the c value and see what happens:

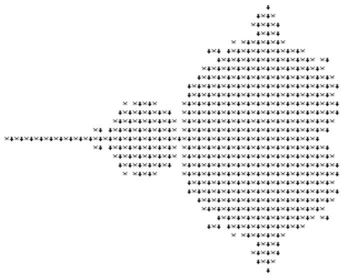


Figure 12.6: Brooks and Matelski's first published picture of the Mandelbrot set. We can do better than that!

Source: Wikimedia Commons

Program 12.3: mandelseries.cpp

```
#include <stdio.h>
#include <math.h>

int main () {
  double c=1;
  double z=0;
  int i;

  printf ("For c = %lf:\n", c );
  for ( i=0; i<10; i++ ) {
    printf ( "z %d = %lf\n", i, z );
    z = pow(z,2) + c;
  }
}
```

The output of this program would be:

```
For c = 1.000000:
z 0 = 0.000000
z 1 = 1.000000
z 2 = 2.000000
z 3 = 5.000000
z 4 = 26.000000
z 5 = 677.000000
z 6 = 458330.000000
z 7 = 210066388901.000000
z 8 = 44127887745906175377408.000000
z 9 = 1947270476915296285689291011464375055838871552.000000
```

Wow! We were right. The numbers get big pretty quickly. But is this true for all values of c? Let's try a negative number and see what happens. How about $c = -1$?:

```
For c = -1.000000:
z 0 = 0.000000
z 1 = -1.000000
z 2 = 0.000000
z 3 = -1.000000
z 4 = 0.000000
z 5 = -1.000000
z 6 = 0.000000
```

```
z 7 = -1.000000
z 8 = 0.000000
z 9 = -1.000000
```

Hmm. It just oscillates back and forth between zero and one, and never gets any bigger. Is this true for all negative numbers? No, if we try $c = -2$ we'll find that, after one flip, all of the rest of the values are equal to 2! If we use $c = -3$, though, we'll see that the numbers once again blow up, and become very large very quickly. This is intriguing![2]

What if we extended this to complex numbers? Would they be even weirder? Yes, indeed they would!

In 1978 two mathematicians, Robert W. Brooks and Peter Matelski, tried this and discovered what's known today as the "Mandelbrot Set". It has manyfascinating qualities, including the fact that its boundary is infinitely rough. If you zoom in on most common-or-garden-variety shapes, you'll find that sooner or later you just see smooth surfaces or curves. Not so with the Mandelbrot set. This shape is equally squiggly at every scale. The Mandelbrot Set's boundary is so squiggly that it behaves as something more than a 1-dimensional curve. We call such shapes "fractals", because they appear to have "fractional" dimensionality.

Approximate fractal shapes often appear in nature. This piece of broccoli is a good example.

Source: Wikimedia Commons

Figure 12.6 shows Brooks and Matelski's first illustration of the Mandelbrot set. It shows the "complex plane", where complex numbers are plotted with their real part on the x axis and their imaginary part on the y axis. This graph uses ASCII characters to indicate the c values on the complex plane which *don't* cause the series to blow up. (We call these c values "stable".) This graph was produced in 1978, when computer technology was much less capable than it is now. We should be able to make a much better illustration using the computers available to us in the 21^{st} century.

Frost patterns on a window also exhibit fractal behavior.

Source: Wikimedia Commons

Program 12.4 is the result. It uses the `Complex` variables we defined in the preceding section. It uses a header file named `complex.h` which contains the `typedef` statement and function definitions we wrote earlier for dealing with complex numbers.

The program divides the complex plane into a 250 × 250 grid. Each point on the grid represents a complex number, c, that we'll test to see if it makes the "Mandelbrot series[3]" blow up. The program takes advantage of something mathematicians have proven about the Mandelbrot series: if a complex number has a magnitude greater than 2, we

[2] If you're a mathematician.

[3] See Program 12.3.

know for sure that it will cause the series to blow up. This means we only need to look at a region within a distance of 2 from the origin. As soon as our series wanders outside of this region, we we know that it will blow up.

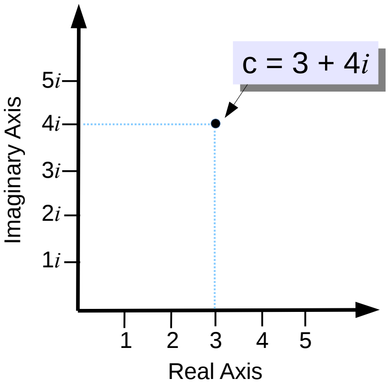


Figure 12.7: The complex number $c = 3 + 4i$ located on the complex plane.

The heart of the program is the function `mandel_test` which we'll use to test each value of c. It calculates up to 100 terms of the Mandelbrot series for this value. If one of the terms wanders more than a distance of 2 away from the origin, we know this c value isn't stable, and we can stop calculating terms. If we make it all the way to 100 terms without becoming unstable, we assume that c is stable. The function just returns the number of terms before instability was detected, or 100 if we made it through all 100.

Each time we test a value of c, we write out its real and imaginary parts, and the number of terms returned by the `mandel_test` function, into a file named `mandel.dat`.

The resulting file will have three columns of numbers: Real part, imaginary part, and number of terms. We can ask *gnuplot* to read this file and interpet it as an image, with the first two columns giving the coordinates of each pixel, and the third column giving its color. We do this by giving *gnuplot* the command "`plot "mandel.dat" with image`". You can see the result in Figure 12.8. Beautiful!

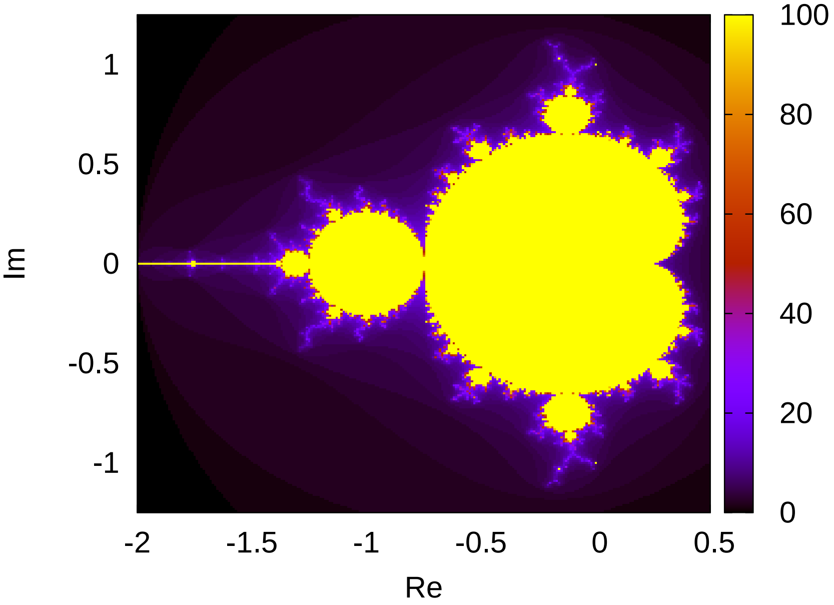


Figure 12.8: The Mandelbrot set, generated by Program 12.4 and visualized by *gnuplot*. See Wikipedia for much more information about this fascinating and beautiful structure.

Exercise 59: Fun with Fractals

Create `mandel.cpp` and `complex.h`. Compile and run `mandel.cpp`, then plot your results with *gnuplot*.

Now try modifying the program by changing the x and y limits in `main`. Make x go from -0.76 to -0.75, and y go from 0.04 to 0.06. This will zoom in on the edge of the circle on the left-hand side of the graph. The result should look like Figure 12.9.

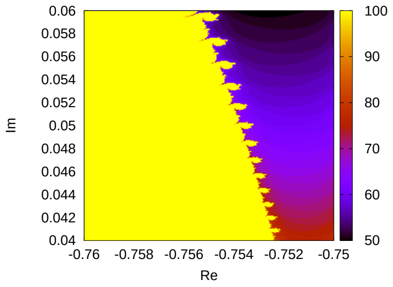
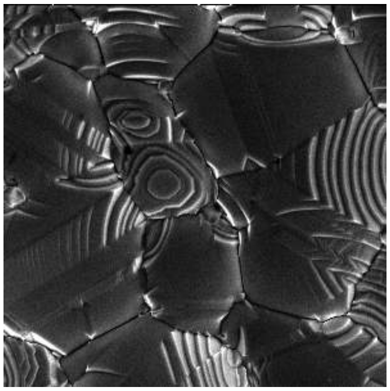


Figure 12.9: Zooming in on the edge of the large circle on the left-hand side of the graph.

12.8. Growing Domains

Let's briefly go away from structures and look at a problem that just uses good ol' `double` values.

Imagine that we have a just-plowed 100 × 100 field of barren earth. Over time, a few seeds of two different species are blown onto the field. The seeds germinate, grow, and begin to reproduce, spreading each species outward from the sites where the initial seeds fell. The two species are incompatible, so new seeds won't grow in land already occupied by the other species.

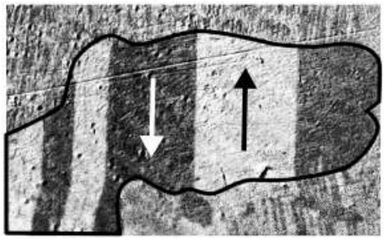


Silver crystals growing on a ceramic substrate. Note the different crystal domains that grow and bump into each other.

Source: Wikimedia Commons

This kind of problem is very common in science. It doesn't have to be the seeds of plants we're talking about. It could be two different crystal structures crystallizing out of a solution, it could be magnetic domains growing in a magnet, or it could be the expansion of human settlements in a formerly unoccupied territory.

Program 12.5 simulates the situation we've described. It defines the 2-dimensional array `color[100][100]` that will hold a color for each sqare area of the field. The color tells us which species is living in that area. The colors will just be three numbers: 0 means the square is empty, 1 means that species number 1 has colonized this area, and 2 means the same for species number 2.



Magnetic domains in a piece of steel.

Source: Wikimedia Commons

The program uses the `rand01` function we wrote in Chapter 9. At the beginning, the elements of `color` are initialized to one of the three colors, based on random "dice rolls" made using `rand01`.

In the middle of the program, we start looping through a large number (1 million) of "turns". During each turn, the program uses `rand01`

Program 12.4: mandel.cpp

```
#include <stdio.h>
#include <math.h>

const int NTRIALS=100;

#include "complex.h"

int mandel_test( Complex c ) {
  Complex z = c;
  int counts = 0;
  while ( magnitude_complex( z ) <= 2.0
&& counts<NTRIALS ) {
    counts++;
    // z -> z^2 + c
    z = add_complex( multiply_complex(z,z), c );
  }
  return counts;
}

int main(){
  double xmin = -2.0;
  double xmax = 0.5;
  double ymin = -1.25;
  double ymax = 1.25;
  Complex c;
  int nim,nre, counts;
  const int NSTEPS = 250;

  FILE *outp = fopen("mandel.dat","w");

  for (nre=0; nre<NSTEPS ; nre++) { // loop over real axis
    c.re = xmin + (xmax-xmin) * nre/NSTEPS;
    for (nim=0; nim<NSTEPS; nim++) { // loop over imaginary axis
      c.im = ymin + (ymax-ymin) * nim/NSTEPS;
      counts =  mandel_test(c);
      fprintf(outp,"%lf  %lf  %d\n",c.re,c.im,counts);
    }
  }
  fclose(outp);
  return 0;
}
```

to pick a random element of `color`. It then randomly picks another element one space left, right, up, or down from that element. If this second element is empty (that is, its color is 0), the second element's color is set equal to the first element. The first element has "colonized" the second.

To pick a random direction, we make use of a new operator that we haven't seen before. This is C's "ternary" operator. Most of the operators in C, like +, -, *, /, *et cetera*, work on one or two values. The ternary operator is the only operator in C that uses three values. It acts like an abbreviated "`if else`" statement, and is indeed exactly equivalent to this. It's just shorter to write.

The syntax of the ternary operator is:

```
condition ? do this if true : do this if false
```

The statement above is exactly equivalent to:

```
if ( condition ) {
   do this if true
} else {
   do this if false
}
```

At the end of Program 12.5 the elements of `color` are printed out in a particular way. We can think of a 2-dimensional array like `color` as being a matrix of some number of rows and some number of columns. In Program 12.5 we print out the array elements in just this way. Each line of the output corresponds to a row of the matrix, and the number of lines is equal to the number of rows.

If we run the program, directing it output into a file like this "`./domain > domain.dat`", we can plot the results with *gnuplot*. We've written the program's output as a matrix of values, which is different from the kinds of files we've asked *gnuplot* to read before. That's OK, though. We just need to let *gnuplot* know that the file is in this format. Here's how to do that:

```
plot "domain.dat" matrix with image
```

The word `matrix` tells *gnuplot* that the file is in the form of an $n \times m$ matrix with a newline at the end of each row.

If we modified the program so that it just showed us the initial dis-

Program 12.5: domain.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }
  return ( rand()/(1.0+RAND_MAX) );
}

int main () {
  int color[100][100];
  int i,j,n,direction,inew,jnew,t;
  double roll;

  // Initialize:
  for ( i=0; i<100; i++ ) {
    for ( j=0; j<100; j++ ) {
      roll = rand01();
      if ( roll < 0.10 ) {
        color[i][j] = 1;
      } else if ( roll < 0.20 ) {
        color[i][j] = 2;
      } else {
        color[i][j] = 0;
      }
    }
  }

  // Take turns:
  for ( t=0; t<1000000; t++ ) {
    i = 1.0 + 98.0*rand01();
    j = 1.0 + 98.0*rand01();

    rand01() < 0.5 ? direction=0 : direction=1;
    rand01() < 0.5 ? n=0 : n=1;
    if ( direction == 0 ) {
      inew = i-1+2*n;
      jnew = j;
    } else {
      inew = i;
      jnew = j-1+2*n;
    }

    if ( color[inew][jnew] == 0 ) {
      color[inew][jnew] = color[i][j];
    }
  }

  // Write results:
  for ( i=0; i<100; i++ ) {
    for ( j=0; j<100; j++ ) {
      printf ("%d ", color[i][j] );
    }
    printf ("\n");
  }
}
```

tribution of seeds, the output would look like the left-hand graph in Figure 12.10. The unaltered program would show us the distribution of species after 1 million turns. That would look like the right-hand graph Figure 12.10.

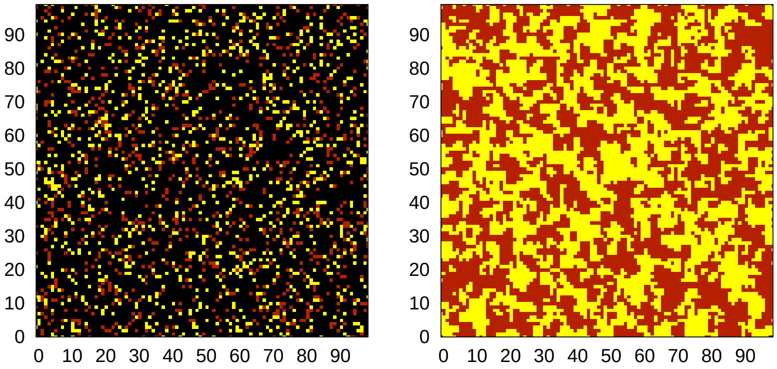
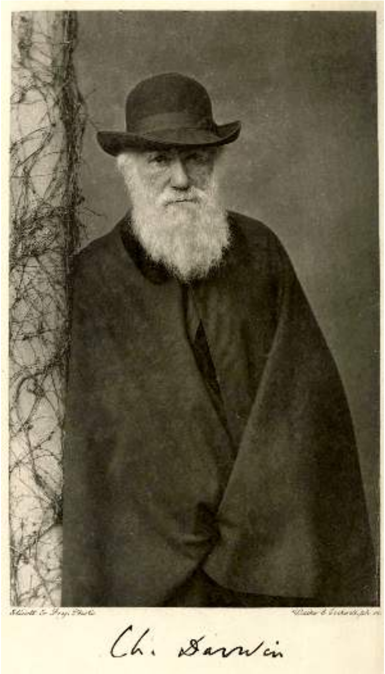


Figure 12.10: Initial distribution of two species in the beginning (left) and after some time has passed (right). Black represents uncolonized spaces.

12.9. Simulating Evolution

In the preceding example, each element of our array has only one property (color). What if the elements had more properties? We could store all of the properties by using an array of structures instead of an array of `ints`.



Charles Darwin, who developed the theory of evolution by natural selection, as described in his 1859 book *The Origin of Species*.

Source: Wikimedia Commons

To illustrate this, let's assume that our two species are small animals that can compete with each other for resources. Each element of our simulation array is a small habitat that can contain a family of these animals, and let's follow the members of each species over a long time and watch them spread and respond to natural selection.

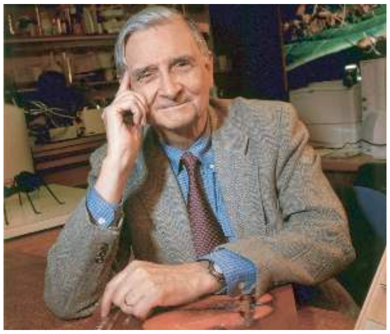
In 1859 Charles Darwin published his *Origin of Species*. Both he and Alfred Russel Wallace had hit, more or less simultaneously, on the idea of "evolution by natural selection". This theory says that evolution occurs because of three factors:

- Inheritance of characteristics. (Individuals tend to pass along some of their characteristics to their offspring.)
- Variability. (Offspring aren't identical to their parents, due to random variations.)
- Natural Selection. (Some characteristics make individuals who possess them more likely to have offspring, either because they make the individual more long-lived, more competitive for resources, more fertile, or through other mechanisms.)

It's not uncommon for larger animals to have an advantage over smaller ones, so let's keep track of the average size of the individuals in each of our small habitats. If the family in one habitat tries to take over an occupied neighboring habitat, we'll assume that the side with bigger individuals will win.

There are also disadvantages to being larger, though. Larger animals tend to reproduce more slowly. This means that it should take more "turns" in our program for larger animals to colonize a new habitat.

A second disadvantage comes from the environment itself. A given area has limited food, water, and other resources. Larger individuals take more resources. If our individuals got too big, they'd be like elephants in a small back yard. There just wouldn't be enough food to support them and they'd eventually die. Mice, on the other hand, could thrive in the same environment.



If you're interested in this kind of thing, you should read a little book called *The Theory of Island Biogeography* by Robert H. MacArthur and Edward O. Wilson (pictured above). If you're even more interested, you should read Wilson's massive tome titled *Sociobiology: The New Synthesis*.

Source: Wikimedia Commons

With these considerations in mind, let's create a structure that could represent each of our array elements. It might look like this:

```
typedef struct {
  int species;  // Which species occupies?
  double size;  // Avg. size of individuals.
  int lastturn; // Last time this family tried expanding.
  double capacity; // Capacity of this habitat.
} Habitat;
```

The structure above records the identity of the species occupying this habitat. This will just be a number: 0, 1, or 2, as in our preceding example. Then it records the average size of the individuals who live in this habitat. As we'll see later, we'll assume that `size` is some measure of the individuals' height. The property `lastturn` records the last time these individuals tried to colonize a neighboring habitat. We'll use this to allow for the fact that it takes larger individuals longer to reproduce. Finally, `capacity` tells us the maximum size of indidvduals that can thrive in this habitat.

We might define a 100 × 100 array of such structures like this:

```
Habitat h[100][100];
```

It'll be convenient to have some functions for dealing with these structures, so let's create a header file that contains these. It might look like "Program" 12.6. You'll see our old friends `rand01` and `normal` from Chapter 9, as well as some new things specific to this program.

The function `init_habitats` sets up the initial conditions by choosing a random species (or no species) for each element of the array. It also sets the size of the inhabitants of this element. It assumes that, intially, the size of all individuals of any species is about "1" (in some arbitrary units), but that there's about a 10% variability between individuals. The function uses our `normal` function to generate the random variations. `init_habitats` also sets the "capacity" of each element to 50. If the size of the individuals in this habitat exceeds this value, bad things will begin to happen for them.

The function `dumpsnapshot` will be used to dump out a "snapshot" of the conditions in our field every once in a while, so we can see how things are progressing. It writes out a file with a name like "`habitat-nn.dat`", where "nn" is number we give `dumpsnapshot`. The file is in the same "matrix" format we used in Program 12.5. The function `meansize` calculates the mean size of the individuals in a given species. As we'll see, this will change as time progresses. This function will let us track those changes.

Program 12.7 uses these structures and functions to actually do our simulation. After calling `init_habitats`, it launches into a loop of 50,000,000 turns. In each turn, the program behaves similarly to Program 12.5. One difference appears in the next-to-last "`if`" statement, which no longer just checks to see if the neighboring element is unoccupied. Now, even if the neighbor is already occupied, it will still be taken over if the its occupants are smaller.

In the final "`if`" statement, we enforce a wait period after we've taken over an element. We're not allowed to take over another element until the wait period has passed. The wait period is calculated from our size. If the size is larger, the wait period is longer (simulating longer gestation periods for larger animals). We assume that the wait is proportional to the mass of individuals. Since we said that `size` was a measure of their height, we assume that their mass is proportional to `size` cubed.

Earlier in the program, after we've picked a random element, we check to see if the size of the individuals in that element has exceeded the element's capacity. If so, we assume they die, and set `species` equal to 0 for that element, making it empty and available for colonization.

When we run the program, it will make two kinds of output. First, it will create ten snapshot files, showing the state of our array at ten different times during its evolution. Second, it will periodically print to the screen two numbers, representing the mean size of species 1 and

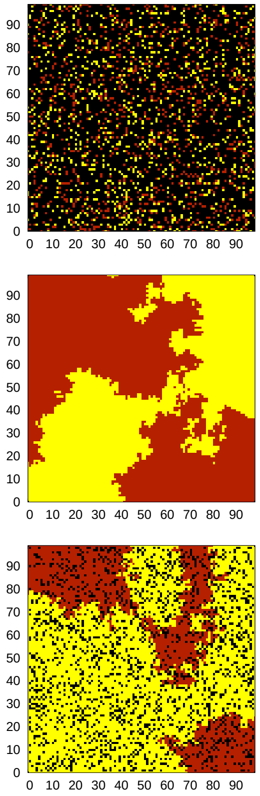


Figure 12.11: Three snapshots in the evolution of our field occupied by two competing species. At the top, we see the initial random distribution. Over time, the families in each of the initial habitats colonize neighboring habitats, perhaps driving out occupants of the other species. The middle snapshot shows an intermediate time, where the species have acheived some kind of equilibrium. Because size is advantageous, natural selection drives members of each species toward larger sizes. In the last snapshot, we see the result when the size of individuals exceeds the capacity of the habitat. Black squares show habitats where colonists have died out due to lack of resources.

When the program calculates `wait` it uses the function `ceil` from C's math library. This function rounds a number up to the nearest integer. There's also a `floor` function, which rounds down to the nearest integer.

Program 12.6: evolve.h

```
double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }
  return ( rand()/(1.0+RAND_MAX) );
}

double normal () {
  int nroll = 12;
  double sum = 0;
  int i;
  for ( i=0; i<nroll; i++ ) {
    sum += rand01();
  }
  return ( sum - 6.0 );
}

double meansize ( int species ) {
  int i,j;
  double sum=0;
  int n=0;
  for ( i=0; i<100; i++ ) {
    for ( j=0; j<100; j++ ) {
      if ( h[i][j].species == species ) {
        sum += h[i][j].size;
        n++;
      }
    }
  }
  return( sum/(double)n );
}

void dumpsnapshot (int isnap) {
  FILE *output;
  char filename[100];
  int i,j;

  sprintf (filename,"habitat-%02d.dat",isnap);
  output = fopen( filename,"w" );
  for ( i=0; i<100; i++ ) {
    for ( j=0; j<100; j++ ) {
      fprintf (output, "%d ", h[i][j].species );
    }
    fprintf (output,"\n");
  }
  fclose( output );
}

void init_habitats () {
  int i, j;
  double roll;
  for ( i=0; i<100; i++ ) {
    for ( j=0; j<100; j++ ) {
      roll = rand01();
      if ( roll < 0.10 ) {
        h[i][j].species = 1;
      } else if ( roll < 0.20 ) {
        h[i][j].species = 2;
      } else {
        h[i][j].species = 0;
      }
      h[i][j].size = 1.0 + variability*normal();
      h[i][j].lastturn = 0;
      h[i][j].capacity = 50.0;
    }
  }
}
```

the mean size of species 2.

We can plot the snapshots with *gnuplot* just as we plotted the output of Program 12.5:

```
plot "habitat-00.dat" matrix with image
```

The result will be plots like the ones shown in Figure 12.11.

If you run the program several times, you'll find that the results will vary widely. Sometimes the two species acheive an equilibrium, as in Figure 12.11, but often one species will completly take over the field.

By directing the program's output into a file ("`./evolve > evolve.dat`") we can look at how the mean size of each species varies over time. Figure 12.12 shows the trend in size for one species in one simulation.

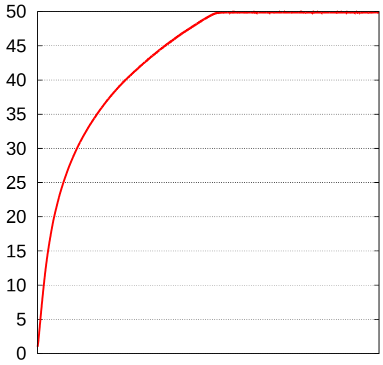


Figure 12.12: Because larger size is favored by natural selection in our model, the mean size of individuals in each species grows over time, until it reaches a plateau at the maximum capacity the habitats can accommodate.

The trend toward larger sizes over time is very common in nature, and is sometimes referred to by evolutionary biologists as "phyletic size increas" or "Cope's Rule" (after palaeontologist Edward Drinker Cope). We're all made familiar with this tendency in childhood, when we first see pictures of tiny early horses like Eohippus (see Figure 12.13).

Figure 12.13: The tiny horse-ancestor, Eohippus, as illustrated by palaeontological artist Charles R. Knight in 1905. Stephen Jay Gould has written an interesting essay about the long history of comparing the size of Eohippus to that of a "fox terrier". You can find it in his collection of essays *Bully for Brontosaurus*.

Source: *Wikimedia Commons*

Program 12.7: evolve.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

typedef struct {
  int species;
  double size;
  int lastturn;
  double capacity;
} Habitat;

Habitat h[100][100];
double variability=0.1;

#include "evolve.h"

int main () {

  int i,j,n,direction,inew,jnew,t,isnap=0;
  double wait;
  int turns=50000000;

  // Initialize:
  init_habitats();

  for ( t=0; t<turns; t++ ) {

    if ( t%(100*100) == 0 ) {
      printf ( "%lf %lf\n", meansize(1), meansize(2) );
    }
    if ( t%(turns/10) == 0 ) {
      dumpsnapshot( isnap );
      isnap++;
    }

    i = 1.0 + 98.0*rand01();
    j = 1.0 + 98.0*rand01();

    if ( h[i][j].size > h[i][j].capacity ) {
      h[i][j].species = 0;
      continue;
    }

    rand01() < 0.5 ? direction=0 : direction=1;
    rand01() < 0.5 ? n=0 : n=1;
    if ( direction == 0 ) {
      inew = i-1+2*n;
      jnew = j;
    } else {
      inew = i;
      jnew = j-1+2*n;
    }
    if ( h[inew][jnew].species == 0 ||
         h[inew][jnew].size < h[i][j].size ) {
      wait = ceil( pow(h[i][j].size,3) );
      if ( t - h[i][j].lastturn > wait ) {
        h[inew][jnew].species = h[i][j].species;
        h[inew][jnew].size = h[i][j].size + variability*normal();
        h[inew][jnew].lastturn = t;
        h[i][j].lastturn = t;
      }
    }
  }
}
```

12.10. Conclusion

Objects in the real world always have more than one interesting property. C's structures allow us to encapsulate an object's multiple properties in a single variable. As we've seen in this chapter, the `typedef` statement can allow us to simplify our code by defining new variable types using these structures.

If we moved forward into the extra features offered by C++, we's see that structures are the precursor of even more powerful things called "classes". A C++ class incorporates multiple properties as well as a set of functions (called "methods") that are particular to a given type of object.

We've also seen several new computing techniques in this chapter. The techniques we used for dealing with gravitational interactions can be refined and improved to make them suitable for really useful calculations of the orbits of celestial bodies. The techniques we've seen for dealing with interactions between neighboring objects (the domain example and the evolution example) have wide applicability in physics and biology.

13. *Bitwise Operators and Binary Numbers*

13.1. Introduction

Back in what this author still regards as "the good old days" it was easy to control individual bits in a computer's memory. Computers like the PDP-11/70 shown in Figure 13.1 actually had switches on the front for doing just that. In order to start the computer, the user would carefully set the switches to a particular pattern of ones and zeroes (usually written on a yellowed piece of paper taped to the front of the computer), perhaps repeating this process several times with different patterns, inching the computer along until it could continue on its own.

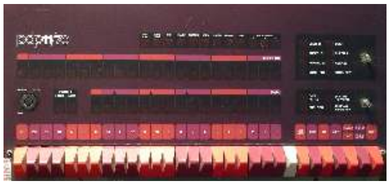


Figure 13.1: The front panel of a DEC PDP-11/70, showing the switches that were used to load a binary starting address into the computer's memory.

Image: Wikimedia Commons

Computers have changed a lot since then, but it's still possible, and sometimes necessary, to switch individual bits on and off. The C programming language provides us with a set of tools for doing that.

It's fun to think about what happens when your program changes the value of a single bit. Each memory cell in a modern computer is smaller than the wavelengths of visible light. When you change the value of a single bit, you're causing a precise physical change in an incredibly tiny object.

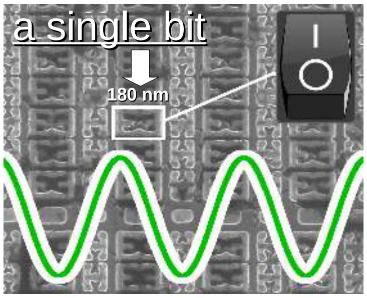


Figure 13.2: An electron microscope's view of a memory chip, showing individual memory cells. Each cell is essentially a little switch, approximately 180 nanometers wide. The wavelength of green light is shown to give an idea of the size.

Image: Wikimedia Commons

Why would you want to the ability to flip individual bits? First of all, bits are the smallest unit of data storage, and by efficiently setting bits you can minimize the amount of space required to store your data on a disk, and the amount of time required to transmit your data from one place to another. Secondly, the CPU in your computer understands how to flip bits on and off, and it can do these operations very quickly. If you can do your calculations by flipping bits instead of more complicated operations like multiplication and division, you can make your program run much faster.

13.2. Binary Numbers

Before we start talking about bits, we first need to understand binary numbers. In our society, we normally write numbers in what might be called "decimal positional" notation. This means that each digit of a number represents some multiple of a power of ten, and the position of the digit indicates which multiple. Take a look at Figure 13.3 for example.

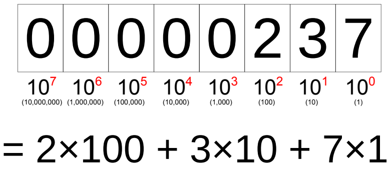


Figure 13.3: The number 237 in decimal positional notation.

The position of each digit tells us how "valuable" it is. You might think of the digits as being like the contents of the bins in a cash register. The rightmost slot is for one-dollar bills, the next is for tens, and the next is for 100s. If we had two $100 bills, three $10 bills, and seven $1 bills, we'd have $237. As we go from right to left, each slot has ten times the value of the preceding one. A number system based on powers of ten is called a "decimal" system from the Greek word *deka*, meaning ten. The `%d` we use when printing integers with `printf` stands for "decimal integer".

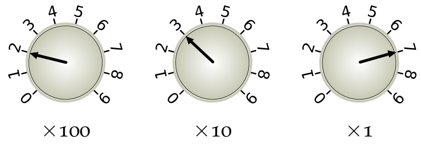


Figure 13.4: To represent a number in our usual decimal notation we need to be able to select from a set of ten digits at each position. We could think of each position as having a dial with ten settings, from zero to nine. The number 237 is shown here.

To make our cash register analogy accurate, we'd have to imagine that as soon as you get ten $1 bills you exchange them for a $10 bill and put that into the next slot to the right, and do a similar operation whenever we get to ten bills in any of the other slots. With this rule, each slot in our number can contain one of ten symbols — 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9 — telling us how many "bills" are in that slot. If we go beyond nine, we need to move to the next slot to the right.

We don't have to use powers of ten, though. We could base our system on any number we want. We probably started using the decimal system because we have ten fingers. If we'd had twelve fingers we might have used a system based on powers of twelve. There are even be some advantages to using such a "duodecimal" system[1]. In fact, vestiges of an old 12-based counting system show up in our daily lives whenever we buy a dozen doughnuts or look at a clock. If we used a 12-based positional system for writing numbers, we'd need twelve possible symbols for each slot.

What if we had to use only two symbols? Then we could write numbers in a "binary" positional notation. (The word binary comes from the Latin *bis*, meaning "twice".) Each slot in a binary number indicates some number of multiples of two, and each digit is either `0` or `1`. (See Figures 13.5 and 13.6.)

Figure 13.5: In binary notation, each slot can contain only a zero or a one, so instead of the knob in Figure 13.4 you might think of each binary digit as a switch.

Why would we be interested in binary numbers? Because each digit

[1] Take a look at this Numberphile video: https://www.youtube.com/watch?v=U6xJfP7-HCc

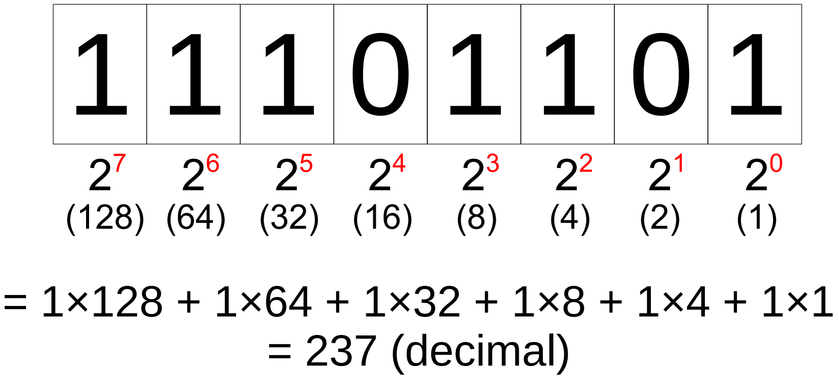


Figure 13.6: The number 237 (decimal) would be written like this in binary.

can be represented by a switch, and it's easy to make switches. We can make switches that are both very small (so that many of them can be packed into a small space) and very fast (meaning that each switch can be turned on or off very quickly)[2]. Once we have some switches, we can use them as the digits of binary numbers.

Each digit in a binary number is called a "bit". You can think of it as a switch that can be flipped to a value of zero or one. Computers usually deal with bits in groups of eight (or multiples of eight). A group of eight bits is called a "byte". (Half a byte, four bits, is sometimes called a "nybble".)

Figure 13.6 shows how the decimal number 237 would be written as a binary number. Each bit represents a power of 2, and can have a value of one or zero. Let's call the right-most bit "bit 0", the next one "bit 1" and so on, with each bit numbered according to the power of 2 that it represents.

There are a couple of things we might notice right away with this system. First, the bit number increases toward the left. If we were given a bunch of bits, numbered zero through seven, and asked to write them down, we might be inclined to start with bit 0 on the left-hand side of the page, then write the others going left-to-right, as we usually arrange things in English. We write the digits of numbers in the opposite way, though, no matter which base (10, 12, 2, or something else) we use. We don't usually think about this, but it's important to keep it in mind as we start working with the digits of binary numbers.

Second, we might notice that this system can only represent positive integers. We haven't provided any way to represent non-integers or even negative integers. We'll address these concerns soon.

Decimal	Binary
1	1
2	10
3	11
8	1000
10	1010
64	1000000
100	1100100
127	1111111
128	10000000
200	11001000
255	11111111

Figure 13.7: Decimal and binary representation of some numbers.

[2] Speed and size are correlated. As switches get smaller, they can also be turned on or off more quickly. That's one reason manufacturers put so much effort into making the already-microscopic components of modern CPUs even smaller.

Figure 13.7 shows the decimal and binary representations of some numbers. Notice that the largest number we can write with eight bits (one byte) is 255. This corresponds to all bits being set to 1. If we want to write larger numbers, we're going to need more bits.

13.3. Bits and Variables

When we write a statement like "number = 42;" in a C program, we're asking the computer to store the value 42 in a variable named number. But what really happens inside the computer? Each variable in our program is just a named section of the computer's memory. When we define a variable named number, the computer reserves a few bits of memory that can be used to store that variable's value.

We can use the sizeof statement[3] (see page 181) to find out how much space has been reserved for a given variable. The space is reported as a number of bytes (8-bit chunks). For example:

```
#include <stdio.h>
int main () {
  int i;
  double d;
  char c;

  printf ("Size of i is %d bytes.\n", (int)sizeof( i ) )
  printf ("Size of d is %d bytes.\n", (int)sizeof( d ) )
  printf ("Size of c is %d bytes.\n", (int)sizeof( c ) )
}
```

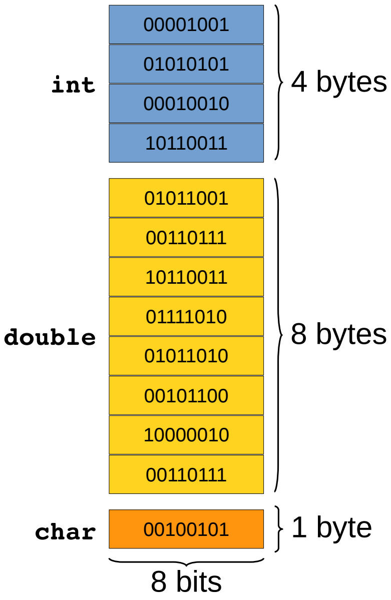


Figure 13.8: Different types of variable use different amounts of storage.

If we compiled and ran this program, we'd see something like this:

```
Size of i is 4 bytes.
Size of d is 8 bytes.
Size of c is 1 bytes.
```

As you can see, different types of variable will generally have different amounts of space. The C language standards don't specify exactly how big the storage space for each type of variable should be, so these numbers may vary from one C compiler to another, but the values shown above are typical.

If the program tells us that int variables are allocated 4 bytes (32 bits) of storage space, what's the biggest number we can store in an int? We might think it would be a binary number with 32 ones, like this:

[3] The value returned by sizeof isn't actually an int, so to keep printf from complaining we force the value to be an int by putting "(int)" in front of it.

```
11111111111111111111111111111111
```

which is 4,294,967,295 (a little over 4 billion) in decimal representation. But if we reached that conclusion we'd be forgetting that `int` variables can hold either positive or negative numbers. Somehow, we've got to reserve at least one bit to indicate whether the number is positive or negative.

It would be great if we could peek at the bits inside an actual variable, to see how they're arranged. Let's start by looking at the smallest kind of variable identified by the program above: the `char` variable, which has only one byte (8 bits) of storage.

13.4. Character/Number Equivalence

You might recall from Chapter 8 that each character is represented by a numerical ASCII code. Figure 13.9 shows the characters corresponding to the ASCII codes from zero to 127. You'll see that the table includes all of the letters and numbers you've come to know and love, and many of the other symbols on your keyboard. For example, the upper case letter "`A`" is character number 65.

There are also some non-printable characters. For example, the NUL character (which we sometimes write as `\0`) is there as character number zero. Character number 10 is the newline (also called linefeed) character, `\n`. Characters like newline, horizontal tab, formfeed, and space are used the move around on your screen, but don't actually print anything. Some of the non-printing characters are extremely weird things like `DLE`, `NAK`, `SYN`, and `ETB`. These are used to control the flow of ASCII data sent across a communications line.

One important fact is that, in C, a character and its ASCII number are almost entirely interchangeable. The following program prints the number 65 twice. The first time, it tells `printf` to print it as a number. The second time, it tells `printf` to print it as a character:

```
#include <stdio.h>
int main () {
  printf ( "As a number it's %d\n", 65 );
  printf ( "As a character it's %c\n", 65 );
}
```

The only substantial difference in the two `printf` lines is that one uses

ASCII		ASCII		ASCII		ASCII	
Number	Character	Number	Character	Number	Character	Number	Character
0	NUL '\0'	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL '\a' (bell)	39	'	71	G	103	g
8	BS '\b' (backspace)	40	(	72	H	104	h
9	HT '\t' (horizontal tab)	41	)	73	I	105	i
10	LF '\n' (new line)	42	*	74	J	106	j
11	VT '\v' (vertical tab)	43	+	75	K	107	k
12	FF '\f' (form feed)	44	,	76	L	108	l
13	CR '\r' (carriage ret)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative ack.)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. blk)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\ '\\'	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

Figure 13.9: ASCII codes between zero and 127 and their corresponding characters.

%d as a placeholder and the other uses %c. The %d tells printf to treat 65 as a number, and %c says to treat it as a character. If we ran the program, we'd see:

```
As a number it's 65
As a character it's A
```

Perhaps even more surprisingly, we'd see the same results if we wrote the program this way:

```
#include <stdio.h>
int main () {
  printf ( "As a number it's %d\n", 'A');
  printf ( "As a character it's %c\n", 'A' );
}
```

As far as C is concerned, 'A' is exactly equivalent to 65.

Exercise 60: Character Building

Write a program named charnum.cpp that uses a for loop to print out the numbers from 33 to 126, inclusive, and the ASCII character that corresponds to each number. The program's output should be two columns, with the first column being the number and the second column its corresponding ASCII character. Note that, because of the equivalence of characters and numbers in C, the loop can either go from 33 to 126 or from '!' to '~' (see Figure 13.9).

If we could look directly at the 8 bits that store a character variable's value, we'd see that they're just a binary representation of a character's ASCII number. For example, 'A' is character number 65, which is 01000001 when expressed as an 8-bit binary number.

13.5. A Simple Encryption Scheme (`rot13`)

Imagine that we took all of the lower-case ASCII letters and arranged them in a circle, as in Figure 13.10. Below each letter is shown its ASCII character number. There are 26 letters, so if we start at any letter then move 13 spaces around the circle we'll find ourselves at a different letter that's exactly on the opposite side of the circle.

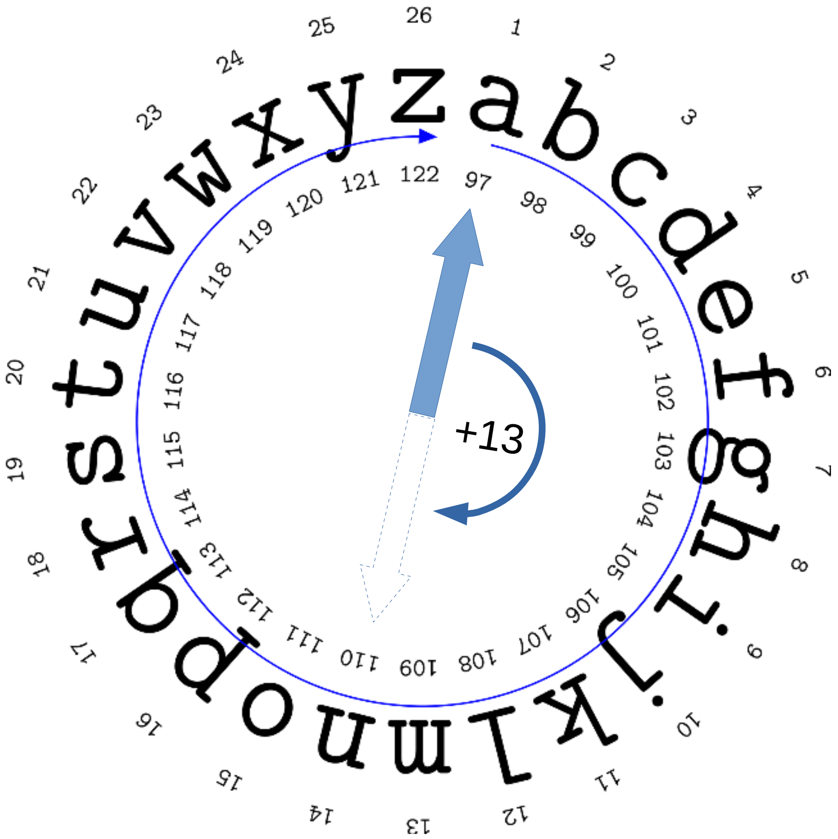


Figure 13.10: Adding 13 to the value `'a'` moves halfway around the circle to `'n'`. Adding 13 again would bring us back to `'a'`.

You could use this as a simple way of "encrypting" a message. Start out by writing down your message, then replace each letter with a different one, halfway around the circle. The person receiving your encrypted message could easily decode it (assuming he or she knows the code!) by just going 13 more spaces around the circle, to get back to the original letter.

This simple encryption scheme is called `rot13`, since it picks a replacement letter by *rotating* 13 spaces around the circle. In the early days of the Internet, `rot13` was often used to obscure text. For example, if you posted a movie review that contained spoilers, you might use `rot13` to encrypt those parts. Anyone who really wanted to read them could decrypt the text, but if you didn't want to know how the movie ended, you'd be in no danger of having it accidentally spoiled for you.

Fortunately, it's easy to write a program that can understand `rot13`. Since moving by 13 spaces can be used to either encrypt or decrypt a message, you can write one program that will work for either task. Give it some plain text, and it will encrypt it. Give it some encrypted text, and it will give you back the original message.

Writing such a program is particularly easy in C, since we're free to use characters and their numbers interchangeably. Program 13.1 shows one way to do it.

Program 13.1: rot13.cpp

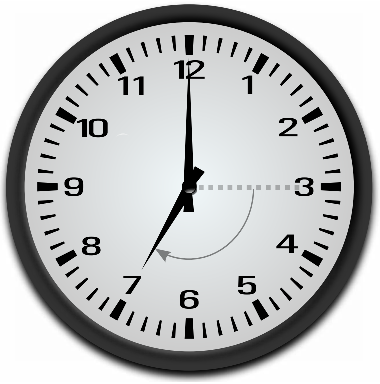
```
#include <stdio.h>
int main () {
  char letter, position, newposition;
  while ( scanf("%c", &letter) != EOF ) {
    if ( letter >= 'a' && letter <= 'z' ) {
      position = letter - 'a';
      newposition = (position + 13)%26;
      letter = 'a' + newposition;
    }
    printf ("%c",letter);
  }
}
```

Ony change lower-case letters.

Shift by 13 letters.

The program uses a "`while`" loop to read characters, one at a time. Each time it reads a character, it checks to see whether this is one of the lower-case characters `'a'` through `'z'`. These are the only letters that are part of the circle that we're using for encryption (see Figure 13.10). Any other characters will be left alone. Notice that we don't have to switch between the character's name (like `'a'`) and its ASCII number (like `97`). C takes care of this for us automatically.

Whenever we find a lower-case letter, we then identify the letter that's opposite it on our letter circle. This circle of letters is a lot like a clock. Remember that in Chapter 4 we talked about clocks, and said that they're an example of modular arithmetic. When a clock's hand goes past twelve, it starts over again at one. We say that the *modulus* of the clock is twelve. If we set a clock's hour hand at 3 and wait 16 hours we'll find that the hand now points to 7. In terms of modular arithmetic, we'd say that `(3 + 16)%12 = 7`, since `7` is the remainder obtained after dividing `3+16` by `12`.



Our circle of letters has 26 positions instead of 12, so it has a modulus of 26. But there's also another difference: our letters don't start at 1. Instead, they start with 'a' (or, equivalently, 97 in C). When we look at a clock, the "12" position is both the beginning and the end of the circle of numbers. When we do modular arithmetic on a clock, we assume that the clock numbers tell us how many hours away from 12 we are. We could imagine that, when the clock's hour hand gets to 12 it's briefly twelve hours away from where it started, then as is passes twelve it's instantaneously back at zero. In our clock's modulo-12 counting system, zero is just the flip side of 12.

So, for example, if we were given the letter 'y' and wanted to find out which letter was on the opposite side of the circle, we first find how far we are from 'a'. This is just 'y'- 'a'. Then we add 13 to this distance and find the remainder after dividing by 26:

```
( 'y' - 'a' + 13 ) % 26
```

The remainder tells us how far away from 'a' we'll be when we move to the letter on the opposite side of the circle. To find out this letter's ASCII number, we just add 'a' to it. That's what Program 13.1 does.

Exercise 61: A Lot of Rot

Create and compile Program 13.1. Run the program and type some text. When you press Enter or Return, the program should print the rot13-encrypted version of your text. For example, if you type *"this is a test"* the program will tell you that the encrypted version of this is *"guvf vf n grfg"*. Press Ctrl-D to exit the program. If you run the program again and type *"guvf vf n grfg"*, the program will translate it back into *"this is a test"*.

If you have a whole file full of text you want to encrypt (a file named secretmessage.txt, for example), you can rot13-encrypt the whole thing by typing this command:

```
cat secretmessage.txt | ./rot13
```

Amaze your friends! Confuse your enemies!

Now that we have some understanding of how character variables are stored, it's natural to wonder how other kinds of variables are stored. It would be nice if we could examine them bit by bit to find out. We can do this, but first we'll need to learn a little about C's "bitwise operators". In particular, we'll need to learn about "bitwise shift" and "bitwise and".

13.6. The Shift Operators

It should be obvious that the following program will print "1". (Try it yourself if you don't believe me!)

```
#include <stdio.h>
int main () {
        printf ( "%d\n", 1 );
}
```

Figure 13.11: If you've never used a typewriter, you might not know that the shift key originally shifted part of the typewriter up or down, to get access to upper-case letters. This reduced the number of keys that were needed. Before the shift key was invented typewriters either had separate keys for each upper and lower case letter, or they could only type in upper (or lower) case.

Image: Wikimedia Commons

but what would the following do?:

```
#include <stdio.h>
int main () {
        printf ( "%d\n", 1<<3 );
}
```

You might be surprised to find that it prints "8". What's going on here? What does that "<<3" do?

Let's think about binary numbers again, and imagine that we have an 8-bit binary number representing the value "1", like this:

0	0	0	0	0	0	0	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
(128)	(64)	(32)	(16)	(8)	(4)	(2)	(1)

Figure 13.12: The number 1 written as an 8-bit binary number. As with decimal numbers, extra zeros on the left-hand side don't matter. We can write 1 or 01 or 00001, and they all mean the same thing.

Referring to Figures 13.12 and 13.13, we see that we could write any power of 2 in binary form by just writing down a lot of zeros and putting a one in the slot corresponding to the desired power. So, 2 (decimal) is written as 10 (binary), 4 is written as 100, 8 is 1000, and so on. If we started with a one in the first slot, we could imagine generating all of the other powers of 2 by just shifting the one to the left by some number of slots. (See Figure 13.13.)

Power	Decimal	Binary
2^0	1	00000001
2^1	2	00000010
2^2	4	00000100
2^3	8	00001000
2^4	16	00010000
2^5	32	00100000
2^6	64	01000000
2^7	128	10000000

Figure 13.13: Powers of 2 written as 8-bit binary numbers.

That's exactly what C's << operator does. It shifts all of the bits in a number toward the left by a given amount. Bits shifted past the left-hand edge are lost, and empty slots on the right-hand side are filled in with zeros. In the program above, 1<<3 means "Start with the number 1 in binary, then shift all of the bits to the left by three spaces." As you can see from Figure 13.14, that would give you 8, and that's what the program above prints out.

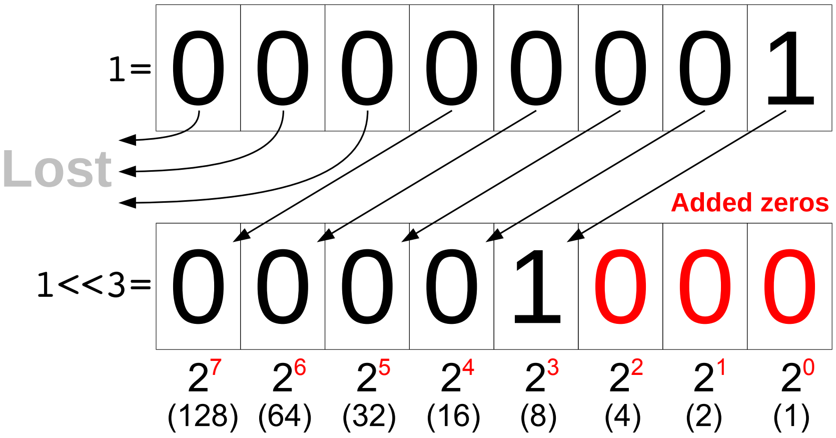


Figure 13.14: Starting with the 8-bit binary representationn of the number `1`, we can shift all of the digits three spaces to the left to get 8 by saying "`1<<3`".

It's interesting to think about what happens to a number when we shift its digits to the left like this. Shifting the digits of a *decimal* number to the left is equivalent to multiplying that number by some power of ten. Ten times 237 is 2370. One hundred times 237 is 23700. Similarly, if we shift the digits of a *binary* number to the left, we multiply it by a power of *two*. For example, `<<1` multiplies the number by two, `<<2` multiplies by four, and `<<3` multiplies by eight.

Figure 13.15: The "left-shift" operator, `<<`, shifts each bit leftward by a given amount. Like birds on a perch, bits that get shifted too far "fall off".

Image: OpenClipart

What do we mean when we say that bits shifted "past the edge" are lost? Where is the edge? As we saw above when we were playing with the `sizeof` statement, each variable in a program has some amount of storage space allocated to it. The bits that represent that variable's value are stored in that space. We can shift those bits around, but the space is finite, and if we shift too far we lose some information[4]. For simplicity, many of the figures in this chapter will assume that we only have eight bits available (one byte), but in reality we'll usualy have more space (four or eight bytes) for each of our numerical variables.

Figure 13.17 shows some examples that start with a different 8-bit binary number (237 in decimal notation). Each time we shift left, some bits drop off the edge and are irretrievably lost. If we shift far enough, as in the bottom case, all of the original bits are lost, and we're left with only zeros. If you only have eight bits to store your number in, you're in trouble if you shove things over by eight spaces.

Note that it's perfectly OK to shift the bits by 0 spaces, even though that doesn't change anything. The expression `1<<0` is just the same as `1`. As we'll see, this is sometimes convenient.

Not surprisingly, there's also a "right-shift" operator, `>>`. Figure 13.18

[4] As we'll see, this also means that there's a maximum number that can be stored in any variable. What that number is will depend on the variable's type.

shows some examples. It works just like the left-shift operator, but moves things in the opposite direction. Don't think that you can use a right-shift to recover bits that have dropped off the edge due to a left-shift, though. That doesn't work[5]. Any bits that are dropped are gone forever.

Exercise 62: Bit Drill

Let's get some practice with the bit-shift operators. Write a program named `bitdrill.cpp` that loops through values of `i` from 0 through 31 and prints `i` and `1<<i` for each value. You might see a surprise for `1<<31` !

When you printed the value of `1<<i` you probably used "`%d`" in your `printf` statement. Try changing this to "`%u`", then recompile your program and run it again. Does the value of `1<<31` change? We'll explain why this happens a little later.

One final note about the exercise above: if you look at Figure 13.13 you can see that the binary representation of each of the numbers you generated (each `1<<i`) would be mostly zeros except for a single `1` in bit number `i`. Apparently, we can use `1<<i` to create a number that just has one particular bit turned "on". This fact will come in handy in the next section.

Okay, so we see that it's possible to shift bits left and right. What good does that do us? Remember that our goal was to be able to see how the bits are really arranged when we store a number in a variable. Bit-shifting is one of the tools we'll need to do that, but we'll also need another tool: the "bitwise *and*". We'll get to that in a later section, but first we need to learn a little more about how a computer stores numbers.

Figure 13.16: Bit drill, meet drill bits.
Image: Wikimedia Commons

[5] More accurately, you can't depend on it.

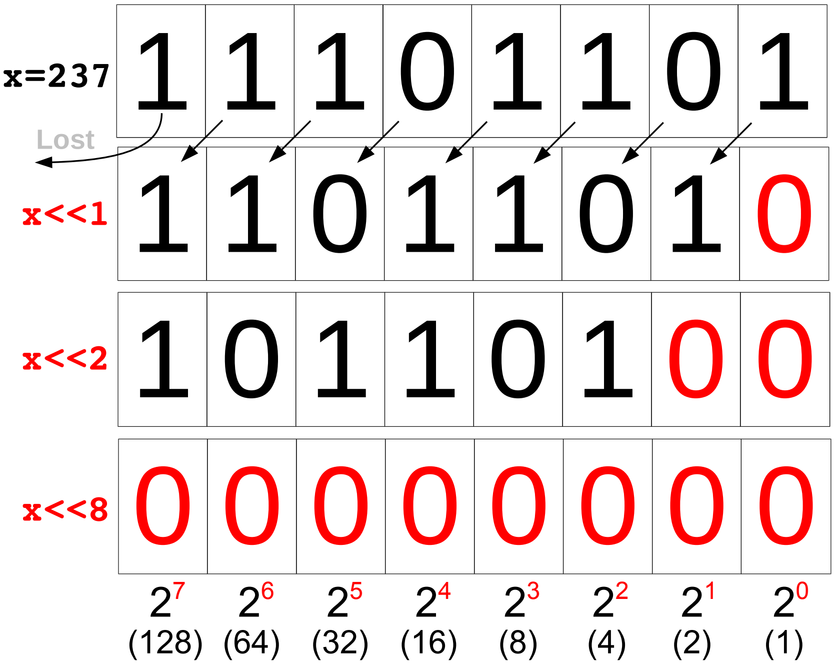


Figure 13.17: If you shift far enough, all of the original bits are lost.

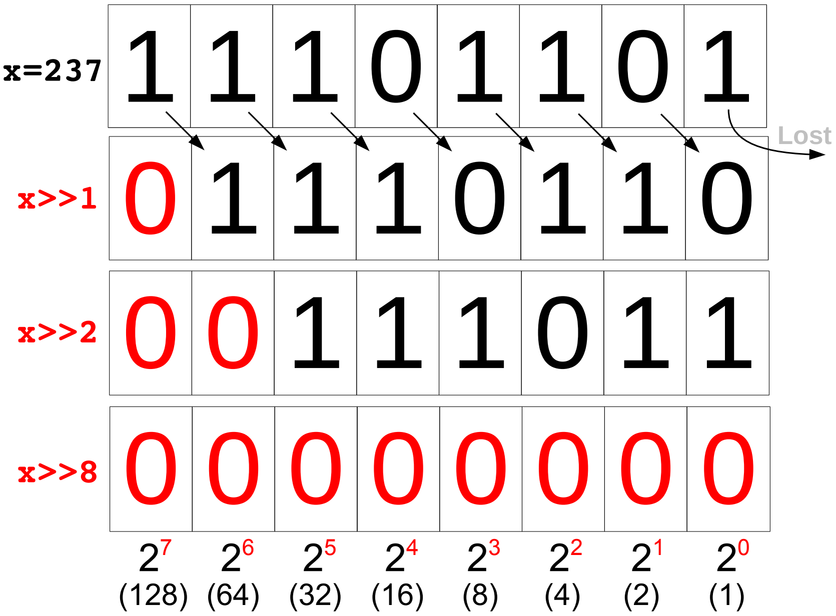


Figure 13.18: The right-shift operator, >>, shifts bits rightward by a given number of slots.

13.7. Signed and Unsigned Integers

In Section 13.6 you might have been surprised by the output of your `bitdrill.cpp` program. If you used "`%d`" when printing the values you probably saw that the program's output looked like Figure 13.19.

Why is the last number negative? To figure it out, let's start by considering what this number's bits look like. Figure 13.20 shows the left-most few bits of this 32-bit number (all of the other bits are zero). We might expect this number to be equal to 2^{31}, which is a little over two billion.

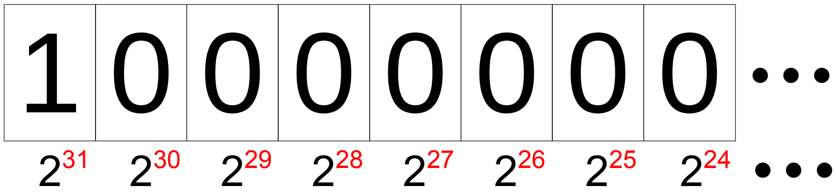


Figure 13.20: The left-most bits of the 32-bit number `1<<31`.

The explanation has to do with the way the computer stores negative numbers. In order to store both negative and positive numbers we need to reserve at least one bit that will indicate the number's sign. That's part of the explanation for what we see in our program's output, but it's clearly not the whole story. If the top-most bit just indicated the sign, then our last number would be "-0", not $-2,147,483,648$.

```
0 1
1 2
2 4
3 8
4 16
5 32
6 64
7 128
8 256
9 512
10 1024
11 2048
12 4096
13 8192
14 16384
15 32768
16 65536
17 131072
18 262144
19 524288
20 1048576
21 2097152
22 4194304
23 8388608
24 16777216
25 33554432
26 67108864
27 134217728
28 268435456
29 536870912
30 1073741824
31 -2147483648
```

Figure 13.19: The output of the `bitdrill.cpp` program from Section 13.6, using `%d` to print the numbers.

To make computations faster, computers actually use a slightly more complicated way of storing negative integers called "two's complement" notation. The two's complement of a binary number can be formed by:

1. Flipping every `1` to a `0`, and every `0` to a `1`, and
2. Adding `1` to the result.

This might seem pointless, but it has a distinct advantage: it lets the computer add numbers together without needing to check their signs. Adding the two's complement of a number turns out to work just the same as *subtracting* that number.

If we just reserved one bit as a "sign bit", we'd always need to check the sign when adding numbers, and then decide whether to add or subtract. This would add extra steps to our calculation, slowing things down. By using two's complement notation we avoid this. As it turns out, the bit pattern we produced by doing `1«31` is the two's complement of $2,147,483,648$, so to the computer it represents the negative of that number.

Using two's complement notation for negative numbers, a 32-bit integer can hold any number between $-2,147,483,648$ and $2,147,483,647$. Any number that has `1` as its left-most bit is assumed to be negative, and interpreted as the two's complement of the value. Figure 13.21 shows the binary representation of some typical numbers.

Decimal	Binary, 32 bits, Signed
0	`00000000.00000000.00000000.00000000`
1	`00000000.00000000.00000000.00000001`
32	`00000000.00000000.00000000.00100000`
256	`00000000.00000000.00000001.00000000`
1 billion	`00111011.10011010.11001010.00000000`
2 billion	`01110111.00110101.10010100.00000000`
2,147,483,647	`01111111.11111111.11111111.11111111`
-2,147,483,648	`10000000.00000000.00000000.00000000`
-256	`11111111.11111111.11111111.00000000`
-32	`11111111.11111111.11111111.11100000`
-1	`11111111.11111111.11111111.11111111`

Figure 13.21: Some representative signed 32-bit integers. Groups of eight bits (1 byte) have been separated by dots for clarity.

The table is arranged in order of increasing binary numbers, from all bits "off" to all bits "on". Notice that when we go past the biggest positive number (a little over 2 billion) the value jumps immediately to the smallest negative number. The value when all bits are "on" is -1.

What if we know that all of our values are going to be positive? Are we still limited to a maximum value of 2,147,483,647? It seems a shame to reserve part of the available range for negative numbers when we know we won't have any.

If you try changing "`%d`" into "`%u`" in your `bitdrill.cpp` program, you'll see that the last thing it prints is now a positive number:

```
31 2147483648
```

The "`%u`" format specifier tells the program to interpret the binary data as an "unsigned integer". Unsigned integers don't wrap around to negative values halfway through their range. Instead, they start at zero and just keep getting bigger. The biggest value that can be stored in a 32-bit unsigned integer is $4,294,967,295$ (a little over 4 billion), which in this case is represented by a 32 "on" bits. Figure 13.23 shows the bit patterns that correspond to some representative unsigned integers.

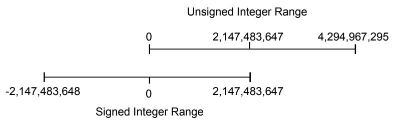


Figure 13.22: A visual comparision of the range of `int` and `unsigned int` variables.

Decimal	Binary, 32 bits, Unsigned
0	`00000000.00000000.00000000.00000000`
1	`00000000.00000000.00000000.00000001`
32	`00000000.00000000.00000000.00100000`
256	`00000000.00000000.00000001.00000000`
1 billion	`00111011.10011010.11001010.00000000`
2 billion	`01110111.00110101.10010100.00000000`
2,147,483,647	`01111111.11111111.11111111.11111111`
2,147,483,648	`10000000.00000000.00000000.00000000`
3 billion	`10110010.11010000.01011110.00000000`
4 billion	`11101110.01101011.00101000.00000000`
4,294,967,295	`11111111.11111111.11111111.11111111`

Figure 13.23: Some representative *un*-signed 32-bit integers. Groups of eight bits (1 byte) have been separated by dots for clarity.

The `int` variables we've used so far are for holding signed integers. If you know you won't need negative numbers, you can define a variable as "`unsigned int`", and use `%u` as a placeholder when reading or writing its value.

Notice that an *unsigned* integer uses the same pattern of bits to represent $4,294,967,295$ as the pattern that's used to represent -1 for *signed* integers. It's worth pausing to think about what this means. If we see 32 "on" bits in the computer's memory, we don't know whether it represents -1 or $4,294,967,295$. It could even represent other values. 32 bits is the same size as four 8-bit `char` variables, so these bits could represent an array of four characters. It's not enough to know what binary data is stored in a variable's memory location. We also need to know the variable's *type*. The type tells us how to interpret the data we see.

Program 13.2 can be used to illustrate the difference between `int` variables and `unsigned int` variables.

Program 13.2: unsigned.cpp

```
#include <stdio.h>
int main () {
  int i;
  unsigned int j;

  printf ( "Enter an integer: " );
  scanf ( "%d", &i );
  printf ( "You entered %d\n", i );

  printf ( "Enter an integer: " );
  scanf ( "%u", &j );
  printf ( "You entered %u\n", j );
}
```

If you ran this program and gave it `4000000000` (4 billion) each time it asked you for a number, you'd see the following:

```
Enter an integer: 4000000000
You entered -294967296
Enter an integer: 4000000000
You entered 4000000000
```

Sometimes we can choose which way we want to display the same data by just changing the placeholder in our `printf` statement, as we did when we changed `%d` to `%u` in the `bitdrill.cpp` program. There are subtle rules that control the way C converts data from one type to another, though, so be careful, especially when comparing variables of different types. Take a look at the following program, for example[6]:

Program 13.3: plusminus.cpp

```
#include <stdio.h>
int main() {
  unsigned int plus_one = 1;
  int minus_one = -1;

  if( plus_one < minus_one ) {
    printf("1 < -1 \n");
  } else {
    printf("Math isn't broken.\n");
  }
}
```

If we compiled this program, *g*++ would give us a warning about comparing signed and unsigned numbers, but it would still create a program we could run. If we ran the program, it would erroneously tell us that 1 is less than -1. That's because in this situation *g*++ assumes we want to compare the two numbers as though they were both unsigned integers. As we saw above, the signed integer -1 is the same as the unsigned integer $4,294,967,295$.

Notice that $4,294,967,295$ is $2^{32} - 1$. In general, if we have n bits for storing an unsigned integer, the biggest number we can store will be $2^n - 1$.

One final note: If you wanted to, you could explicitly define `int` variables as `signed int`, to make clear that they're different from `unsigned int`. You don't have to, though. If you don't specify whether the variable is signed or unsigned, the default is `signed`.

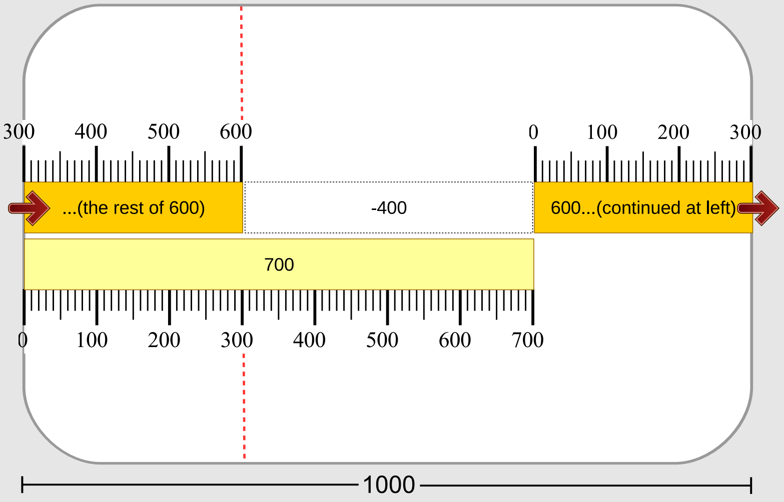
[6] This example is adapted from an excellent, but rather technical, explanation by Ozgur Ozcitak at StackOverflow.

But what about. . . ?

What does “two’s complement” mean, anyway? And how can you possibly subtract by adding? The answers depend on the fact that numbers in a computer always have a limited number of digits.

For example, an `unsigned int` variable might have 32 bits — 32 binary digits – in the computer’s memory. If we try to put too large a number into that space, the upper digits of the number will be lost. If the biggest number we can store is 4,294,967,295 (expressed in binary, of course) but we try to put in 4,294,967,296, we’ll see that our variable ends up containing the value zero! 4,294,967,297 would give us 1, 4,294,967,298 would give us 2, and so forth. It’s like the numbers get to the maximum value and then wrap around to the beginning again.

This is analogous to an old-fashioned arcade game like Asteroids, where characters that went off the right side of the screen reappeared on the left side.



It might be easier to understand if we look at a “ten’s complement” example, where we work in the more-familiar base 10. Imagine that we want to subtract 400 from 700. Let’s start by putting a 700-pixel-long bar on the screen of a 1,000-pixel-wide arcade console, as in the figure above.

If we wanted to subtract 400 pixels from the bar’s length, we could

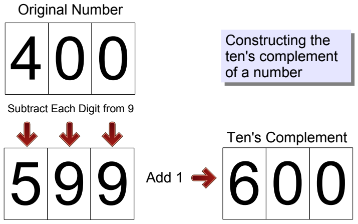

Original Number
4 0 0
Constructing the ten's complement of a number
Subtract Each Digit from 9
5 9 9
Add 1
Ten's Complement
6 0 0

Figure 13.24: The “ten’s complement” of a number can be formed by subtracting each digit from 9 and then adding 1 to the resulting number.

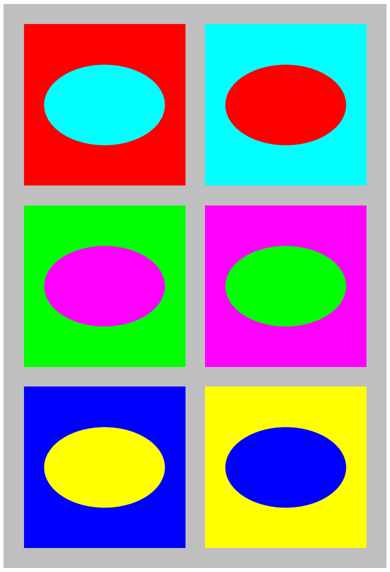


Figure 13.25: “Complementary colors” are pairs of colors that combine to form white (or black in some color systems). Each is the “complement” of the other, meaning that each supplies what the other lacks. Similarly, the ten’s complement (or two’s complement) of a number supplies what the number lacks to become a power of ten (or two).

just move to the left by that amount and chop the bar off at the vertical dashed line. This is the same as subtracting 400 pixels from the bar's length.

Alternatively, though, we could find the ten's complement of 400 and *add* that much to the bar's length, causing it to wrap around when it bumps into the edge of the screen. The ten's complement of the number is just the difference between the number and the total width of the screen. That difference turns out to be 600 pixels in this example. As you can see, going forward a distance of 600 pixels (which wraps us around to the other side of the screen) leaves us at the same dashed vertical line. We end up at the same place as if we'd subtracted 400 pixels.

For 3-digit numbers the ten's complement is the amount you need to add to get to 1,000, because you can only hold numbers up to 999 in three digits. After that, the numbers roll over like an odometer to 000. For 4-digit numbers the limit would be 9,999, and the ten's complement would be the amount you need to add to get 10,000. In general, for n digits the ten's complement of a number x is $10^n - x$. Similarly, the n-digit two's complement of x would be $2^n - x$.

One way to find the ten's complement of a number is to subtract each of the number's digits from 9, and then add 1 to the result, as shown in Figure 13.24. This might seem roundabout when we could just subtract the number from 1,000 (for 3 digits) and be done with it, but it's useful when working with binary numbers. In base 2, instead of subtracting from 9, you just flip the value of each bit. This is something the computer can do very quickly. It turns out that flipping the bits and adding 1 to the result is much faster than finding the twos complement any other way.

Figure 13.26: Many cartoon characters also have four digits. This is Felix the Cat, one of the author's favorites. He first appeared in *Feline Follies* in 1919. You can watch it at archive.org.

Image: Wikimedia Commons

13.8. Bitwise Logic

Way back in Chapter 3 we learned about the "and" and "or" (`&&` and `||`) logical operators that we often use inside "`if`" statements. For example, the statement:

```
if ( a<3 && b>4 )
```

can be read as "if `a` is less than three **and** `b` is greater than four", whereas the statement:

```
if ( c==1 || d<7 )
```

means "if `c` equals one **or** `d` is less than seven".

The `&&` operator compares two expressions and tells us whether *both* expressions are true. The `||` operator compares two expressions and tells us whether *at least one* of them is true.

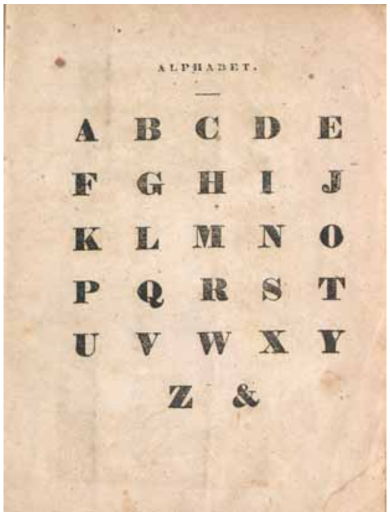
It turns out that C has a set of similar operators for comparing individual bits of binary numbers. These operators are `&` and `|`. (Note that, unlike the logical operators we've used before, these new operators aren't doubled. Each is just a single character.) These operators treat "`1`" as *true* and "`0`" as *false*.

Consider the example in Figure 13.29, which sets `z` equal to `x&y`. As you can see, the `&` operator compares two numbers, bit by bit, looking for places where the bits of both numbers are set to "`1`". If both bits are "`1`", then the resulting bit is "`1`", otherwise, it's "`0`".

We can summarize the behavior of the `&` operator with a *truth table*, as in Figure 13.30. This shows the value that a bit of `x&y` will have if the corresponding bits of `x` and `y` have the values given in the shaded squares. A bit of `x&y` is only true (has a value of "`1`") if the corresponding bits of `x` and `y` are both true.

Figure 13.27: The mathematics of logic is called "Boolean Algebra" after its inventor, George Boole, a 19^{th} Century British mathematician. Boole studied how true and false assertions could be chained together using "ands" and "ors" to trace a mathematically rigorous path leading to a specific conclusion. His work was the foundation of modern computer science.

Image: Wikimedia Commons



ALPHABET.

A B C D E
F G H I J
K L M N O
P Q R S T
U V W X Y
Z &

Figure 13.28: The ampersand, `&`, once had the distinction of being a member of the alphabet, as seen in this page from the 1863 "Dixie Primer, for the Little Folks". When reciting the alphabet the "little folks" would end by saying "X, Y, Z, and *per se* and", the slurring of which gave rise to the character's name. The character itself is a combination of the letters *Et*, the Latin word for "and".

Image: Wikimedia Commons

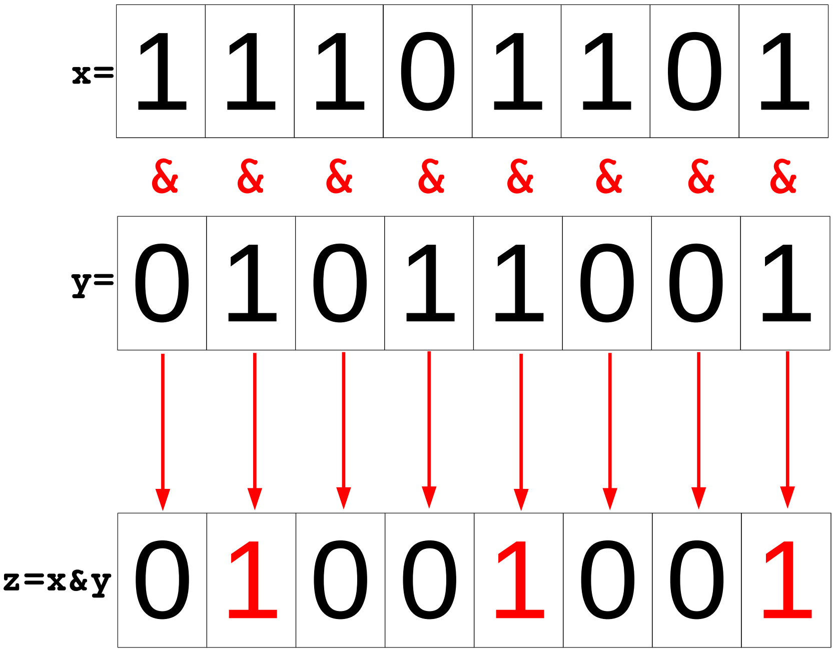
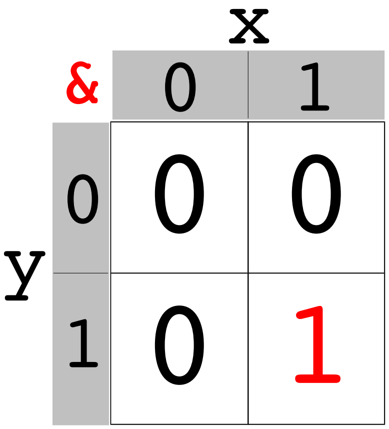


Figure 13.29: The result of a bitwise "and" of two binary numbers is another number that only has a `1` in the spots where *both* of the original numbers had a `1`.



		x	
	&	0	1
y	0	0	0
	1	0	1

Figure 13.30: Truth table for `&`, the "bitwise and" operator. The result is only true if both x and y are true.

The `&` operator is valuable because it can be used to find out whether a particular bit of a given number has a value of `0` or `1`. Let's try it out by examining the bits in the number 237. You can see this number's binary representation at the top of Figure 13.31.

Now let's constuct another binary number. Recall that we can make a binary number with a single `1` in any slot we choose by starting with `1` and shifting with the `<<` operator. The middle line of Figure 13.31 shows a number that has a single `1` in slot number 3. The number is `1<<3`. That's equal to 8 in decimal notation, but all we really care about is the fact that it's all zeros except for a `1` in bit number 3. We might call this number a "mask" because (as we'll see) we're going to use it to hide all but one bit of the first number.

If we "bitwise and" these two numbers together, the result is what's shown in the bottom row of Figure 13.31. The `1` in this row is telling us that bit number 3 is "on" in the number we're testing (the top row).

We can use other masks to test other bits. For example, Figure 13.32 shows how we could use `1<<4` as a mask to test bit number 4 of our number. In this case, the bottom row shows all zeros, indicating that bit number 4 is "off".

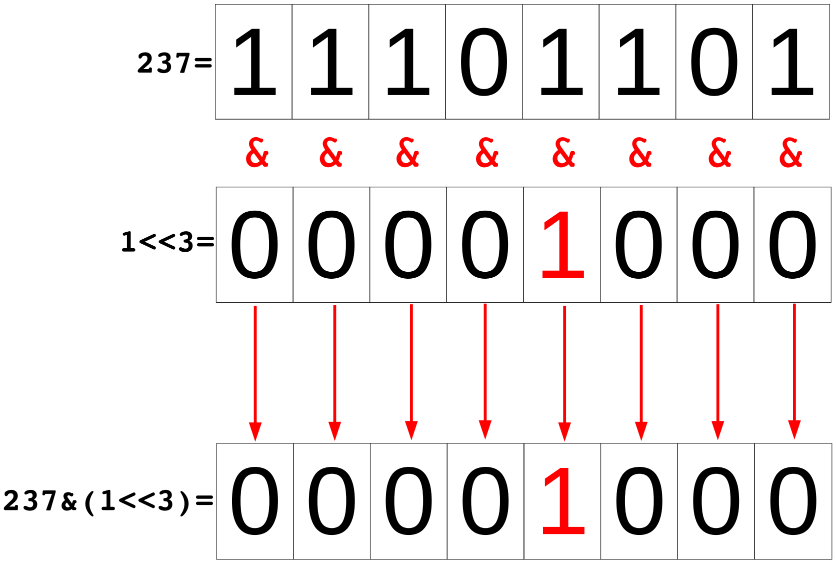


Figure 13.31: Testing bit 3 of the number 237. The result shows that this bit is "on".

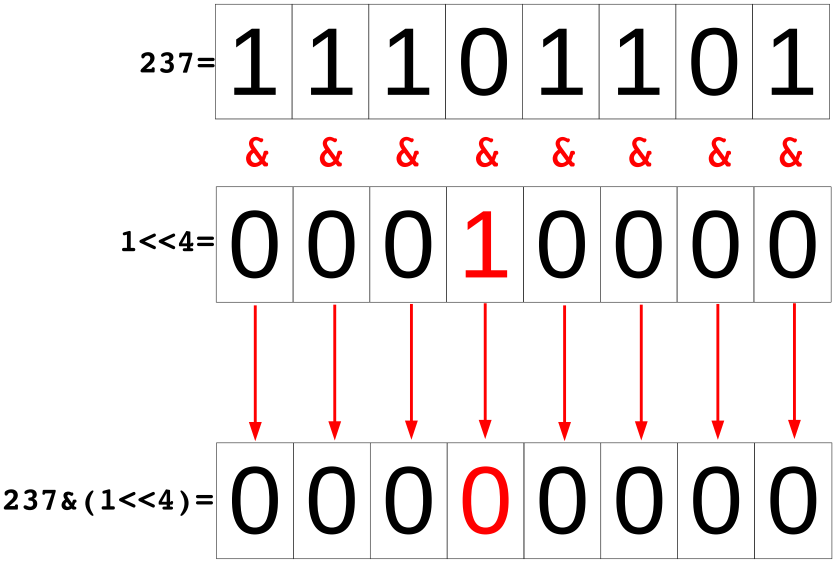


Figure 13.32: Testing bit 4 of the number 237. The result shows that this bit is "off".

All of this leads us to the general rule that we can test bit number i of a number, n, by checking the value of n&(1<<i). If this value is zero, then the bit is "off". If the value is non-zero, then the bit is "on". We'll use this fact in the next section, when we start examining the inner workings of variables.

Before we go on, though, let's look at another useful bitwise logic operator: the "bitwise or" operator, |. Figure 13.33 shows how this operator works. If we have two values, x and y, and combine them with the |operator to get a new value, z = x|y, the result has a 1 in any slot where either x or y had a 1. The truth table for this operator is shown in Figure 13.34.

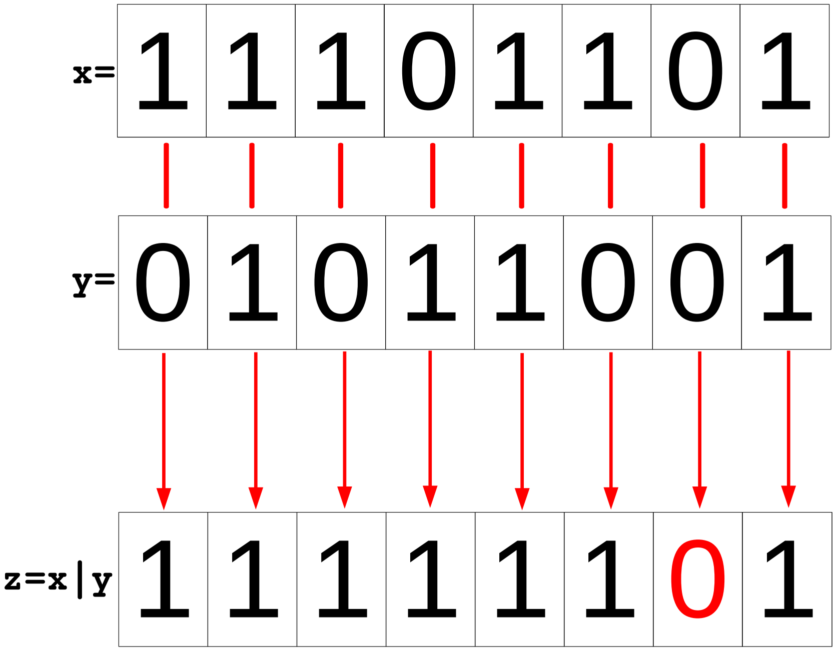
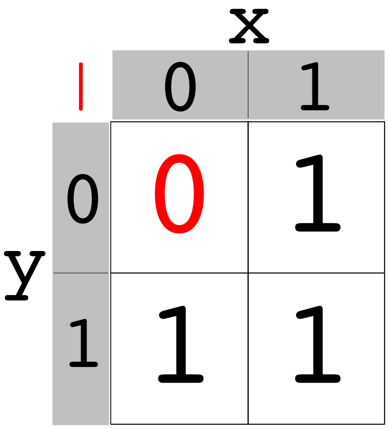


Figure 13.33: The result of a bitwise "or" of two binary numbers is another number that has a 1 in the spots where *either* of the original numbers had a 1.



|	x=0	x=1
y=0	0	1
y=1	1	1

Figure 13.34: Truth table for |, the "bitwise or" operator. The result is true wherever either x or y is true.

Another bitwise operator we should talk about is the "exclusive or" (often called "xor") operator, ^. Unlike the "or" operator, z = x^y gives a 1 in any position where one *and only one* of the original numbers has a 1. Figure 13.35 illustrates this. As you can see in the bottom row, there's a zero wherever both bits are "off", but there are also zeros whenever both bits are "on". The truth table for the ^operator is shown in Figure 13.36.

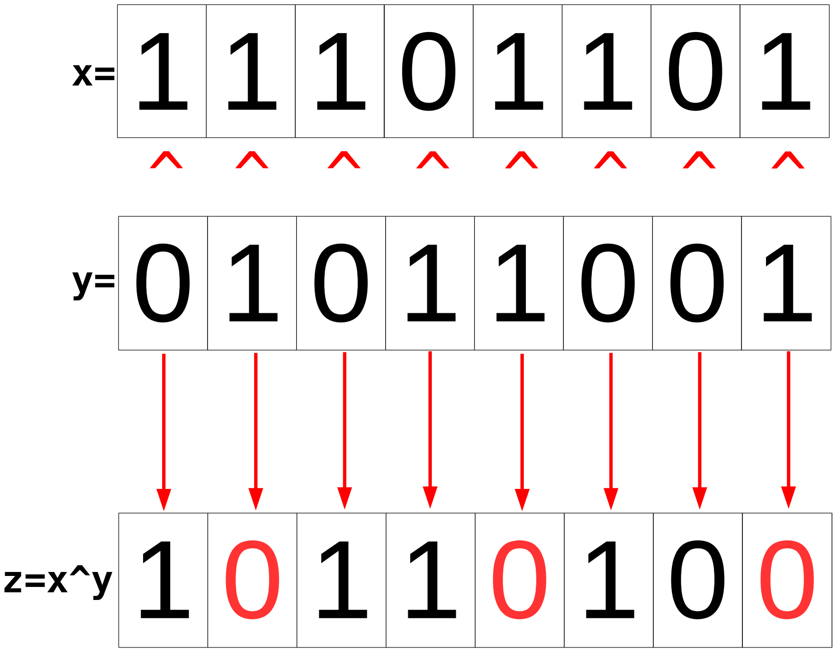
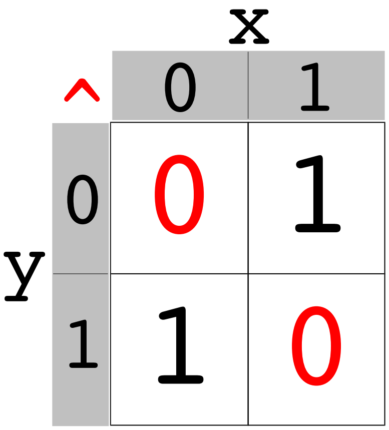


Figure 13.35: The result of a bitwise "exclusive or" of two binary numbers is another number that has a 1 in the spots where *only one* of the original numbers had a 1.



^	x = 0	x = 1
y = 0	0	1
y = 1	1	0

Figure 13.36: Truth table for ^, the "exclusive or" operator. The result is true wherever only one of x or y is true.

Finally, to complete our toolkit of bitwise operators there's the "bitwise not", ~. This operator changes every 0 to 1 and every 1 to 0. For example, if x contains the bits 11101101, then ~x will be 00010010. If you think of each 1 or 0 as "true" or "false", then the "not" operator changes each "true" into "not true", and each "false" into "not false".

The following table summarizes the bitwise logical operators we've talked about in this section:

Operator	Symbol	Usage	Description
Bitwise **and**	&	z = x&y	Bits of z are 1 only where bits of both x and y are 1.
Bitwise **or**	|	z = x|y	Bits of z are 1 where bits of either x or y are 1.
Bitwise **xor** ("Exclusive or")	^	z = x^y	Bits of z are 1 where bits of either x or y are 1, but not where both are 1.
Bitwise **not**	~	z = ~x	Bits of z are the opposite of the corresponding bits in x.

13.9. Examining Bits

Now that we know about bit-shifting and the bitwise "and" operator we're ready to write a program that examines individual bits in the computer's memory. Remember that we can check the value of `n&(1<<i)` to find out whether bit number `i` of the number `n` is "on" or "off", as we saw in Section 13.8.

Program 13.4 puts the value 42 into a variable named `n` and then examine's the variable's bits one by one.

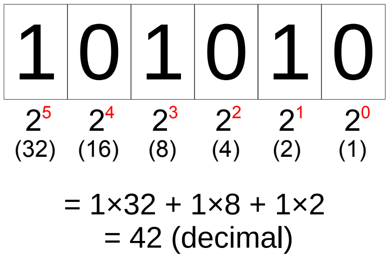


Figure 13.37: The number 42 expressed in binary.

Program 13.4: printbits.cpp

```
#include <stdio.h>
int main () {
  unsigned int n = 42;
  int i, on, nbits;

  nbits = 8*sizeof( n );

  for ( i=nbits-1; i>=0; i-- ) {

    on = n & (1<<i);

    if ( on != 0 ) {
      printf ( "1" );
    } else {
      printf ( "0" );
    }

  }

  printf ("\n");
}
```

`sizeof` gives bytes.
1 byte = 8 bits.

Start at highest bit and work down.

If this value is zero, the bit is "off".

If we ran this program, it would print:

```
00000000000000000000000000101010
```

Is this really equal to 42 in decimal? Let's take a look. We can ignore the long line of zeros on the left-hand side, so we only need to look at the right-most 6 bits. As you can see from Figure 13.37, this is indeed the binary representation of the decimal number 42.

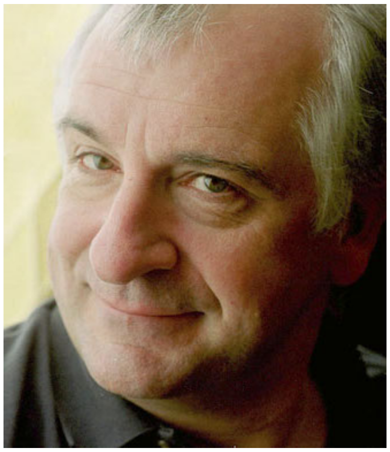


Figure 13.38: In Douglas Adams' *Hitchhiker's Guide to the Galaxy* books, 42 is the answer to the "ultimate question of life, the universe and everything".

Image: Wikimedia Commons

Exercise 63: Bit by Bit

Create, compile and run Program 13.4. It should print the binary version of the decimal number 42. Try changing the value of `n` in the program, recompiling it and running it again. What happens if you set `n` to a power of 2 (like 2, 4, 8, 16, and so forth)? What happens if you set it to a value that's one less than a power of 2 (like 3, 7, 15, 31, ...)?

Try a few even numbers, paying attention to the right-most digit of the output, and then try a few odd numbers doing the same. Do you see a pattern?

If you're tired of re-compiling the program, modify it so that it asks you for the number instead of having the number written into the program. Pay attention to the kind of format specifier (placeholder) you use in your `scanf` statement. Make sure it matches the type of the variable you're reading the number into.

Try giving the program the number 4294967295 (the biggest number that an `unsigned int` can hold. What does the output look like? What happens when you give it even bigger numbers?

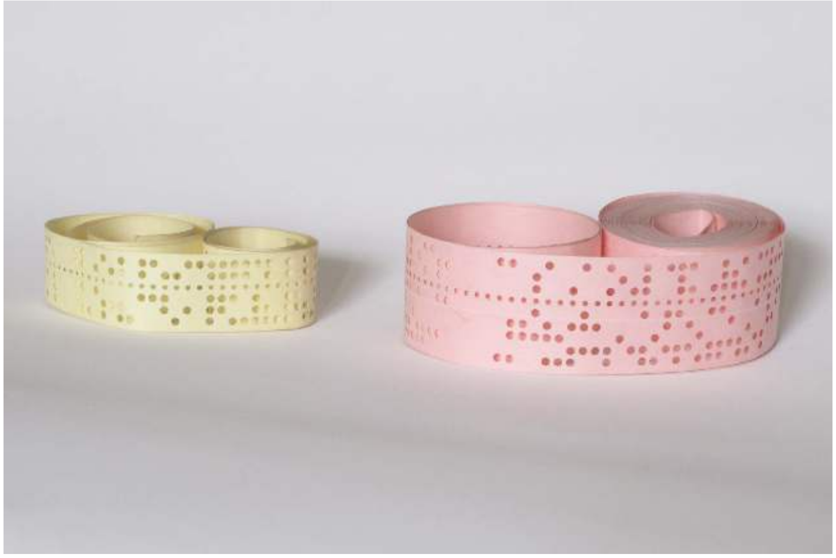


Figure 13.39: In the past, data was sometimes saved on paper tapes like this. Each line (vertical column in this picture) represents a binary number. A large punched-out hole represents a `1` and the un-punched spaces are zeros. (Ignore the line of small holes. That's for moving the tape.) The right-hand tape in this picture is written using 7-bit ASCII characters. The bottom-most position in each line is a special "parity bit" wich isn't part of the character but is used for error-checking. The other bits can be read from bottom to top as a binary number representing an ASCII character (see the table in Figure 13.9). The visible part of the right-hand tape says: `10.1 TYPE"DO YOU LOVE ME?"` Let's hope the poor programmer wasn't disappointed.

Image: Wikimedia Commons

13.10. Using xor for Encryption

You and your best friend want to exchange secret messages. Fortunately, you know about the bitwise "exclusive or" (xor) operator, ^, and that's all you need for writing a simple encryption program.

Take a look at Program 13.5 (`secretletter.cpp`) below.

Figure 13.40: *Les Deux Soers* (1889), by Pierre-Auguste Renoir.

Image: Wikimedia Commons

Program 13.5: secretletter.cpp

```
#include <stdio.h>
#include <stdlib.h>
int main ( int argc, char *argv[] ) {
  char letter;
  char key;

  if ( argc != 2 ) {
    fprintf ( stderr, "Usage: %s key\n", argv[0] );
    exit(1);
  }

  key = argv[1][0];

  while ( scanf( "%c", &letter ) != EOF ) {
    printf ( "%c", letter^key );
  }
}
```

Make sure user has supplied a key.

The key is the first (and only) letter of the first argument.

Combine each letter with the key, using xor.

This program takes one command-line argument: a single letter that forms the "key" for your encrypted message. Only someone who knows the key will be able to unscramble the message.

The program works by taking each letter you type and "xor-ing" its bits with the bits of the key. The resulting encrypted letter is then printed out. The program will keep reading letters until it sees an "End of File" signal, which you can give it by typing Ctrl-D. The encrypted characters the program generates might not be viewable on your screen, and some of them might even cause your display to misbehave in weird ways. Because of this, it would be a good idea to redirect the program's output into a file, like this:

```
./secretletter b > secretstuff.dat
```

In this example I've used the letter "b" as my secret key, but you can use any character you like. Just don't tell anyone except your friend.

Once you've made the encrypted file (`secretstuff.dat` in the example above), you can e-mail it to your friend, secure in the knowledge that nobody else will be able to read it.

But how will your friend decode the message? That's where the xor operation really comes in handy. It turns out that if you have three binary numbers, `plain`, `key`, and `encrypted`, and you do this to them:

```
encrypted = plain ^ key
```

then the following is *also* true:

```
plain = encrypted ^ key
```

so all your friend needs to do is run the encrypted message back through the same program, using the same key. Once your friend receives your message, he or she can decrypt it by typing:

```
cat secretstuff.dat | ./secretletter b
```

This is like what we did earlier to "decrypt" files created with Program 13.1 (`rot13.cpp`).

One obvious weakness of this program is that a Bad Guy could decrypt our message by just trying all the possible letters we might have used as our secret key. Even if we allow any letter (upper or lower case) or number as the key, that's still only 62 possibilities, and it wouldn't take that long to try them all.

We could improve our security by using a whole word as our key – a password! Each time the program encrypts a character it could use the next letter in the word, until it gets to the end and then starts over at the beginning of the word. With that change, the number of possible keys (still assuming only letters and numbers) becomes 62^n, where n is the maximum length of our password. For 8-letter passwords, that's $62^8 = 218,340,105,584,896$ possiblities! It would be very hard for a Bad Guy to try all of these.

Program 13.6 shows how you might modify the `secretletter.cpp` program to make it use a whole word as the key. The necessary changes are shown in bold. Notice that we use the `strlen` function to find the length of the "key word" (or "password"), and we use the modulo operator (%) and the number of characters read so far (`nchars`) to keep cycling through the letters of the key word.

Figure 13.41: Julia Child — author of *The Art of French Cooking* and beloved host of *The French Chef* — had an earlier career as a spy. During World War II she worked for the OSS, stationed in Sri Lanka and China, where she passed encrypted intelligence back to the US.

Image: Wikimedia Commons

Program 13.6: secretword.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main ( int argc, char *argv[] ) {
  char letter;
  char key;
  int keynumber, keylength, nchars=0;

  if ( argc != 2 ) {
    fprintf ( stderr, "Usage: %s key\n", argv[0] );
    exit(1);
  }

  keylength = strlen( argv[1] );

  while ( scanf( "%c", &letter ) != EOF ) {
    keynumber = nchars%keylength;
    key = argv[1][keynumber];
    printf ( "%c", letter^key );
    nchars++;
  }
}
```

Figure 13.42: *"Say the secret word and win a hundred dollars!"* Groucho Marx was the host of a quiz show named *You Bet Your Life*. At the beginning of the show the audience was shown a secret word. If a contestant used the word during the quiz, a rubber duck holding a $100 bill descended on a string.

Image: Wikimedia Commons

This program works just the same as the earlier version. Start typing your message after entering a command like this:

```
./secretword groucho > secretstuff.dat
```

where "`groucho`" is whatever you choose to use as your password and `secretstuff.dat` is the encrypted version of your message. Again, type Ctrl-D when you're finished typing the message. Your friend can then decrypt the message by typing:

```
cat secretstuff.dat | ./secretword groucho
```

The simplicity of xor encryption comes with a price. Another relation between the key, the plain message, and its encrypted version is this:

```
key = encrypted ^ plain
```

meaning that, if an enemy ever obtains both the encrypted and decrypted versions of one of your messages, they can find the key you've used! Even with a multi-letter key, if bad guys ever get snippets of encrypted and unencrypted data that are longer than our key, they'll be able to calculate all the letters in the key. Because of this weakness, the same password shouldn't be used more than once when doing this kind of encryption.

In the days of the cold war, Soviet spies came to the US armed with a pad full of encryption keys. Their associates back in the USSR had identical pads. Whenever a spy needed to send back some information, he'd use one of the keys to encrypt it, then throw away that key. When his compatriot received the message, he'd decrypt it using the first key on his pad, and then discard that key. This type of encryption key is called a "one-time pad".

Figure 13.43: A Soviet poster: *"In order to have more, it is necessary to produce more. In order to produce more, it is necessary to know more."*

Image: Wikimedia Commons

Exercise 64: Spies Like Us

Create, compile, and run Program 13.6 (`secretword.cpp`). Try encrypting a message, writing the encrypted output into a file named "`encrypted.dat`".

Look at `encrypted.dat` with *nano*. It should look like nonsense.

Now try decrypting the message. Does the text displayed on your screen match your original message? What happens if you use the wrong password when attempting to decrypt the message?

What happens if you only use *the first part* of the password when decrypting the message? For example, if you used "charlottesville" as the password when encrypting the message, what happens if you use "charlotte" when decrypting it? (Make sure your message is longer than the password.)

Note that you can use spaces in the password, but if you do you'll need to enclose it in quotes, like this:

```
./secretword "a long password" > encrypted.dat
```

and do the same when decrypting the message.

13.11. `long` and `long long` Variables

There are almost 8 billion people on Earth. Imagine that you had to give each person an ID number. You wouldn't be able to store that number in an `int` or even an `unsigned int`. They can only store numbers up to about 2 and 4 billion, respectively.

What if we want to store an integer that's bigger than the biggest thing that will fit into an `unsigned int`? C offers some other variable types that might accommodate your needs. Two of them are "`long int`" and "`long long int`".

The C standard doesn't specify how many bits each of these types has. It only requires that `long int` be *at least as* large as `int`, and that `long long int` be *at least as* large as `long int`. In some cases, two (or even all three) of these types of variables will have the same number of bits. Typically, though, you'll find that at `long long int` can hold significantly larger numbers than `int`.

Figure 13.44: Long, long hair! An average human has about 100,000 scalp hairs, a number that could easily be stored in an `int`.

Image: Wikimedia Commons

We can again use `sizeof` to find out how many bits each of these types uses.

```
#include <stdio.h>
int main () {
  int i;
  long int ilong;
  long long int ilonglong;

  printf ("Size of i is %d bytes.\n", (int)sizeof( i ) );
  printf ("Size of ilong is %d bytes.\n", (int)sizeof( ilong ) );
  printf ("Size of ilonglong is %d bytes.\n", (int)sizeof( ilonglong ) );
}
```

If we ran this program on a typical computer, we might see something like this:

```
Size of i is 4 bytes.
Size of ilong is 8 bytes.
Size of ilonglong is 8 bytes.
```

Since a byte is 8 bits, this means that an `int` has $4 \times 8 = 32$ bits, a `long int` has $8 \times 8 = 64$ bits, and a `long long int` also has 64 bits. If we stored the number 42 in an `int` variable on this computer, its bits would look like this:

00000000000000000000000000**101010**

but the same value stored in a `long int` would look like this:

0000000000000000000000000000000000000000000000000000000000**101010**

Just as with plain `int` variables, there are signed and unsigned versions of `long` and `long long int` variables: `unsigned long int` and `unsigned long long int`.

Each of these types has an appropriate placeholder to use in `printf` and `scanf` statements. The table below lists them.

Type	Format Specifier
`int`	`%d`
`unsigned int`	`%u`
`long int`	`%ld`
`unsigned long int`	`%lu`
`long long int`	`%lld`
`unsigned long long int`	`%llu`

What's the biggest number that will fit into each of these types of integer? With the information from `sizeof` and what we now know about how integers are stored we could figure it out, but there's an easier way. Your programs can use the file `limits.h` to find the biggest or smallest numbers we can put into a given type of variable. `limits.h` defines symbols like `INT_MAX` and `INT_MIN`, which tell you the maximum and minimum valuse an `int` can hold. Here are some useful values to look at:

Symbol	Meaning
`INT_MIN`	The smallest negative value that can be held in an `int`.
`INT_MAX`	The largest positive value that can be held in an `int`.
`LONG_MIN`	The smallest negative value that can be held in a `long int`.
`LONG_MAX`	The largest positive value that can be held in a `long int`.
`LLONG_MIN`	The smallest negative value that can be held in a `long long int`.
`LLONG_MAX`	The largest positive value that can be held in a `long long int`.
`UINT_MAX`	The largest value that can be held in an `unsigned int`.
`ULONG_MAX`	The largest value that can be held in an `unsigned long int`.
`ULLONG_MAX`	The largest value that can be held in an `unsigned long long int`.

Notice that there are no symbols for the minimum values of the unsigned types. For those, the minimum is always zero.

With all that in mind, we could write a little program to tell us the limits on our various integer types. Program 13.7 does that.

Program 13.7: printsizes.cpp

```
#include <stdio.h>
#include <limits.h>
int main () {
  printf ("INT_MAX is %d\n", INT_MAX );
  printf ("LONG_MAX is %ld\n", LONG_MAX );
  printf ("LLONG_MAX is %lld\n", LLONG_MAX );

  printf ("INT_MIN is %d\n", INT_MIN );
  printf ("LONG_MIN is %ld\n", LONG_MIN );
  printf ("LLONG_MIN is %lld\n", LLONG_MIN );

  printf ("UINT_MAX is %u\n", UINT_MAX );
  printf ("ULONG_MAX is %lu\n", ULONG_MAX );
  printf ("ULLONG_MAX is %llu\n", ULLONG_MAX );
}
```

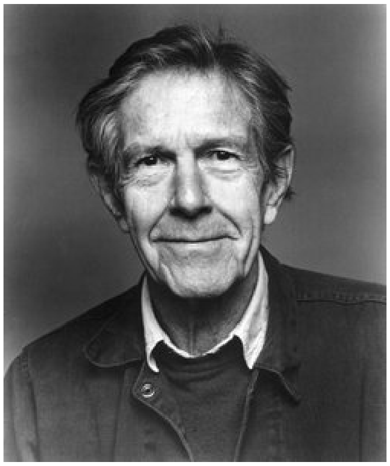


Figure 13.45: Composer John Cage. The longest piece of music I'm aware of is John Cage's *Organ^2/ASLSP (As Slow as Possible)*, which is currently being performed on an organ in Halberstadt, Germany. The performance will end on September 5, 2640, after 639 years! If we created a timer that counted how many seconds the performance has lasted, it would only need to count to a little over 20 billion. This wouldn't fit into an `int`, but it would easily fit into a 64-bit `long long int`.

Image: Wikimedia Commons

If we ran this program on a typical computer we might see something like the following:

```
INT_MAX is 2147483647
LONG_MAX is 9223372036854775807
LLONG_MAX is 9223372036854775807
INT_MIN is -2147483648
LONG_MIN is -9223372036854775808
LLONG_MIN is -9223372036854775808
UINT_MAX is 4294967295
ULONG_MAX is 18446744073709551615
ULLONG_MAX is 18446744073709551615
```

This tells us that, on this computer, the biggest number we can store in any of these integer types is $2^{64} - 1$, or $18,446,744,073,709,551,615$ (about 1.8×10^{19}, or 18 quintillion). This is the maximum value of an `unsigned long int` or an `unsigned long long int` here.

How big is 18 quintillion? That's more than twice the estimated number of grains of sand on earth! We could give each sand grain a serial number if we wanted to.

Figure 13.46: Mathematicians at the University of Hawaii have estimated that there are about 7.5 quintillion grains of sand on Earth.

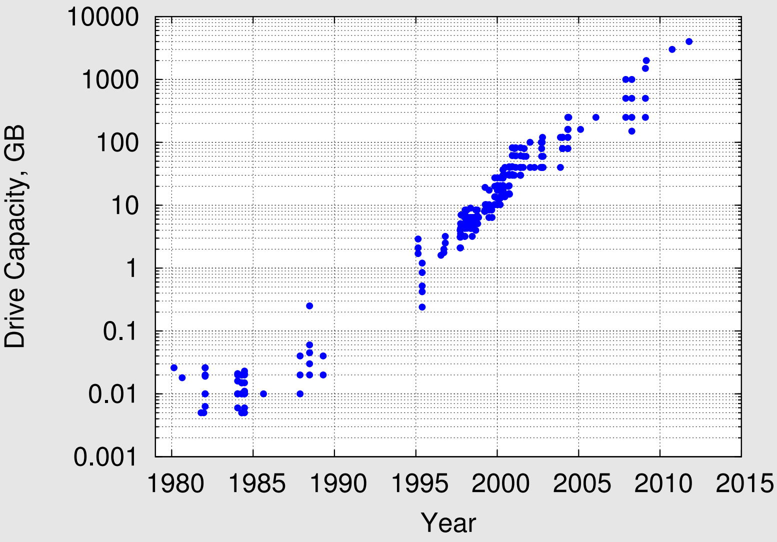
Image: Wikimedia Commons

But what about...?

Operating systems like Microsoft Windows and computer processors like those made by Intel often say they're "64-bit" or "32-bit". What does that mean?

64-bit processors are CPUs that can read and process data in 64-bit chunks. This effectively lets them do more work in less time than a 32-bit processor. Most CPUs you'll find in desktop and laptop computers today are 64-bit devices.

A 64-bit operating system takes advantage of a 64-bit CPU's abilities. One key advantage of 64-bit operating systems is that they can store the addresses of memory or disk locations in 64-bit-wide variables. For example, a 32-bit operating system has trouble with files larger than 2 Gigabytes or amounts of memory larger than 4 Gigabytes — the maximum numbers that can be stored in signed and unsigned 32-bit integers. As we've seen above, the limits on 64-bit integers are astronomically higher, allowing 64-bit operating systems to use much larger files and amounts of memory.



Data adapted from Wikimedia Commons

As you can see from the graph above, disks have continued to grow rapidly in size over the last 40 years. Disks holding tens of thousands of Gigabytes are now available. The move to 64-bit operating systems was necessary to accommodate this growth.

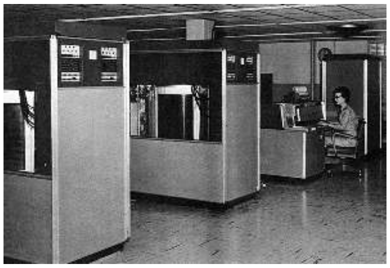


Figure 13.47: The two closet-sized boxes in the foreground are IBM 350 disk drives, introduced in 1956. Each drive can hold a whopping 3.75 Megabytes of data. That's about the size of a single photograph from a modern digital camera.

Image: *Wikimedia Commons*

13.12. int Variables with Specific Widths

As we noted above, the C standards don't specify the exact number of bits that an int, long, or long long variable should have. The standards just say that each has at least as many bits as the preceding type.

Figure 13.48: Two uintatheres pause for a drink in this beautiful painting by Charles R. Knight.
Image: Field Museum

Sometimes, though, we want to have an integer variable with a specific number of bits. For example, if we wanted our program to read binary data in 4-byte chunks, it would be nice to have a 4-byte-long variable to store each chunk.

The header file stdint.h defines some new types that we can use in situations like this:

Type	Size	Format for printf	Format for scanf
uint8_t	8 bits (1 byte)	PRIu8	SCNu8
uint16_t	16 bits (2 bytes)	PRIu16	SCNu16
uint32_t	32 bits (4 bytes)	PRIu32	SCNu32
uint64_t	64 bits (8 bytes)	PRIu64	SCNu64

These variable types each hold unsigned integers. There are corresponding signed types with names like int8_t, but you'll probably find that the unsigned types are more useful.

There are new placeholders ("format specifiers") for each of the new types, and that these placeholders look different from the ones we've used before. In order to use them, you'll first need to include inttypes.h[7]. Notice that there are different placeholders for printf and scanf.

These placeholders are used a little differently, too. Take a look at the following example, which reads a number into a uint8_t variable and then prints the value back out:

Program 13.8: uintathere.cpp

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>
int main () {
  uint8_t n;
  printf ("Enter a number: ");
  scanf( "%"SCNu8, &n );
  printf ("You entered %"PRIu8"\n", n);
}
```

[7] Some older C compilers will complain about these formats unless you also add this arcane line at the top of your program:
`#define __STDC_FORMAT_MACROS`

Instead of saying something like "%d", where the placeholder is inside the quotes, these new placeholders need to go *outside* the quotes. Notice, for example, how the template we give printf in the program above has three sections: a beginning quoted section, followed by PRIu8, then finally another quoted section. Whenever you want to use one of these new-fangled placeholders, you need to exclude it from the quotes like this, but *leave the % inside!*

Here's another example, where we're printing two uint8_t variables:

```
printf ( "The numbers are %"PRIu8" and %"PRIu8"\n", n, m );
```

Why all these complications? It's because these types were added to the C standards long after the original types like int. The new types add new functionality without breaking anything that's already there. This required a little fancy footwork.

Figure 13.49: The scientist and author Carl Sagan was famous for talking about the "billions and billons" of stars out there. The phrase was used so often that an informal unit called the "sagan" has been defined. It's equal to at least 2 billion plus 2 billion ("billions and billions"), or 4 billion.

Image: Wikimedia Commons

13.13. The Size of Literal Numbers

When a program does a calculation like "n = 2*5-3" it stores the numbers 2 and 5 somewhere in the computer's memory, then multiplies 2*5 and stores the result somewhere, then adds 3 to the result. What amount of memory does the program reserve for these numbers and intermediate results while it's working?

In general, the program will look at each number and try to find an appropriately-sized type of storage to put it in. For example, the current version of *g++* will first try to treat the number as an int. If it won't fit into the number of bits allocated for an int, it will move up to a long int or a long long int. The numbers 2, 5, and 3 in the example above would all fit into an int, which has 32 bits (4 bytes) on most computers. If we looked at the computer's memory while the program was running, we'd see something like this:

```
00000000.00000000.00000000.00000010
00000000.00000000.00000000.00000101
00000000.00000000.00000000.00000011
```

representing (from top to bottom) 2, 5, and 3. (I've inserted dots in the numbers above to separate them into byte-sized chunks for clarity.)

If we used the number "8000000000" (8 billion) in our program, it might be stored in memory like this:

```
00000000.00000000.00000000.00000001.11011100.11010110.01010000.00000000
```

since that number is too large to fit into only 32 bits.

You can use the `sizeof` statement to find out exactly how many bytes a program will use to store a given number. For example, this program:

```
#include <stdio.h>
int main () {
  printf ( "%d bytes\n", (int)sizeof(2) );
  printf ( "%d bytes\n", (int)sizeof(2000000000) ); // 2 billion.
  printf ( "%d bytes\n", (int)sizeof(4000000000) ); // 4 billion.
  printf ( "%d bytes\n", (int)sizeof(8000000000) ); // 8 billion.
}
```

might print:

```
4 bytes
4 bytes
8 bytes
8 bytes
```

That's all interesting, but do we need to worry about it? Yes, it turns out that we do sometimes. Imagine what would happen if we had a statement like:

```
n = 2000000000*4+7;
```

which starts by multiplying 2 billion (a number that will fit in 4 bytes) by 4.

When the computer sees an expression like `2000000000*4` it guesses how much space to allocate for the result by looking at the sizes of the numbers being multiplied. Since each of the numbers in this example would fit into 4 bytes, the computer allocates 4 bytes for the result. But, as we've seen above, the result here (8 billion) won't fit into 4 bytes.

Most modern compilers are smart enough to anticipate this problem, and they'll give you a warning message like:

```
warning: integer overflow in expression
n = 2000000000*4+7;
               ^
```

But the compiler can only catch the most obvious variations on this problem. Imagine what would happen in a slightly more complicated situation:

Program 13.9: literal.cpp

```
#include <stdio.h>
int main () {
  int x;
  long int n;

  printf ( "Enter multiplier: " );
  scanf ( "%d", &x);

  n = 2000000000*x+7;

  printf ( "%ld\n", n );
}
```

Here, instead of multiplying by 4, we ask the user to enter a multiplier when we run the program. The compiler can't know in advance what the user will type, so the compiler would give you no warning or error messages, but if you ran the program and entered 4 as the multiplier, bad things could happen.

The variable `n` is a `long int`, which is large enough[8] to hold the expected result of our calculation (8,000,000,007), but the program might incorrectly tell us that the answer is -589,934,585!

What's happening in this case? Let's follow the process step by step:

1. The program multiples 2,000,000,000 × 4, which should equal 8 billion. If we had 8 bytes (64 bits) to store the number, it would look like this:

   ```
   00000000.00000000.00000000.00000001.11011100.11010110.01010000.00000000
   ```

2. Since the computer has only allocated 4 bytes, the left-most `1` gets chopped off:

   Chop this off

   ```
   00000000.00000000.00000000.00000001. 11011100.11010110.01010000.00000000
   ```

   leaving us with:

   ```
   11011100.11010110.01010000.00000000
   ```

   This isn't equal to 8 billion now. Interpreted as an `int`, it's equal to -589,934,592.

3. We now add 7 to this, to get -589,934,585, which is exactly what the program told us.

[8] See Section 13.11 above. Note that the details will vary, depending what operating system and compiler you use, but the principles are the same everywhere.

How could we have avoided this problem? It turns out that you can tell the compiler how much space to reserve for a number. In the example above, we could fix the problem by adding one letter:

```
  n = 2000000000L*x+7;
```

The `L` after the number tells the compiler that we want to reserve as many bits as a `long int` variable[9]. With that change, the program would correctly tell us that the answer is 8,000,000,007.

We can use other suffixes on numbers to select other types. The table below shows some of the possibilities:

Type	Suffix
`long int`	`L`
`long long int`	`LL`
`unsigned int`	`U`
`unsigned long int`	`UL`
`unsigned long long int`	`ULL`

If you want to use the specific-width types like `uint32_t`, defined in `stdint.h`[10], the syntax is a little different. For those types, you can use a statement like:

```
n = UINT64_C(2000000000)*x+7;
```

which tells the compiler that you specifically want to reserve 64 bits for storing the first number. Some other similar options are listed in the table below:

Type	Syntax
`uint8_t`	`UINT8_C()`
`uint32_t`	`UINT32_C()`
`uint64_t`	`UINT64_C()`

As you can see, numbers inside a computer aren't as simple as the numbers we use in math class. The complications arise because the computer has a limited amount of space to store each number. You should think about this whenever you're working near the limit of the largest numbers your variables can contain.

[9] We could have used either an upper- or lower-case `L`. I've used an upper-case letter here to avoid mistaking it for the number `1`.

[10] See Section 13.12.

13.14. Hexadecimal Numbers

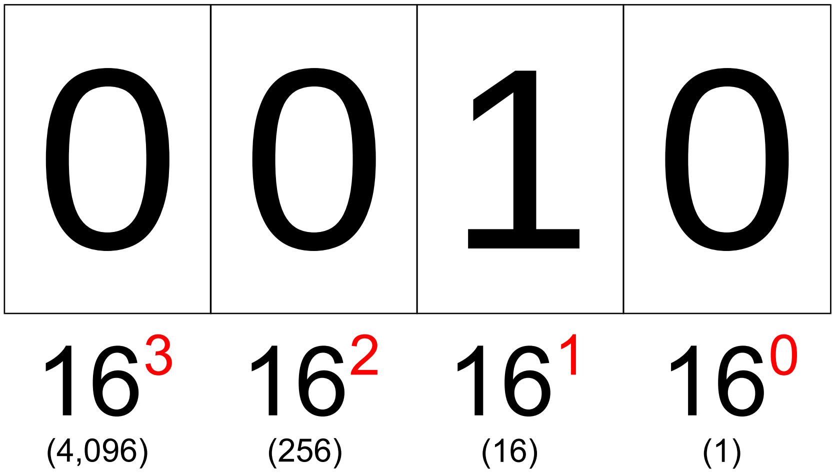
When we write a number like `1729` we assume that it's expressed in base 10 (decimal) notation. In the preceding sections we've seen that it's also possible to write numbers in base 2 (binary) notation. There are a couple of other useful notations that you should be aware of. One of them is "hexadecimal" (or "hex"), which uses 16 as its base.

Figure 13.50: Srinivasa Ramanujan (1887-1920) was a brilliant Indian mathematician. While visiting Ramanujan, the English mathematician G.H. Hardy remarked that the number on a taxicab, 1729, was rather uninteresting. Ramanujan replied that this number was, in fact, *very* interesting, being the smallest number that's the sum of two cubes in two different ways:$1^3 + 12^3$ and $9^3 + 10^3$. Since then, 1729 has been known as "Ramanujan's taxicab number".

Image: Wikimedia Commons

We know that in base 10 we have ten digits (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) and in base 2 we have two digits (0 and 1). Apparently we'll need sixteen digits for base 16! Since we only have ten number symbols on our keyboards, what symbols do we use for the extra six digits? The convention is to use the letters 'A' through 'F'. The table in Figure 13.51 shows some decimal numbers with their hex equivalents. It also shows the binary version of each number.

As you can see, this is how we'd count to sixteen in hexadecimal: 1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,10. In hex, the number "10" is equal to $1 \times 16 + 0 \times 1$:



Hexadecimal numbers are useful whenever you can divide a set of bits into 4-bit groups. Notice that in table in Figure 13.51 we've split the binary version of each number into two 4-bit chunks. You might recall that 4 bits (half a byte) is called a "nybble". The minimum number that can be stored in 4 bits is, of course, zero, and the maximum number is 15. That gives 16 possible values that we can represent with 4 bits, just like the 16 possible digits of hexadecimal numbers.

Decimal	Hex	Binary
0	0	`0000 0000`
1	1	`0000 0001`
2	2	`0000 0010`
3	3	`0000 0011`
4	4	`0000 0100`
5	5	`0000 0101`
6	6	`0000 0110`
7	7	`0000 0111`
8	8	`0000 1000`
9	9	`0000 1001`
10	A	`0000 1010`
11	B	`0000 1011`
12	C	`0000 1100`
13	D	`0000 1101`
14	E	`0000 1110`
15	F	`0000 1111`
16	10	`0001 0000`
17	11	`0001 0001`
18	12	`0001 0010`
31	1F	`0001 1111`
32	20	`0010 0000`
63	3F	`0011 1111`
64	40	`0100 0000`
100	64	`0110 0100`
128	80	`1000 0000`
255	FF	`1111 1111`

Figure 13.51: Some decimal numbers with their hexadecimal and binary (8-bit) equivalents. A space splits each binary number into two 4-bit "nybbles".

Since there are two nybbles in each byte, that means that the value stored in a byte can be represented by two hexadecimal digits. That value can be anything from `00` through `FF`, which corresponds to a range of decimal values from zero to 255.

Let's look at the binary representation of the number 1729:

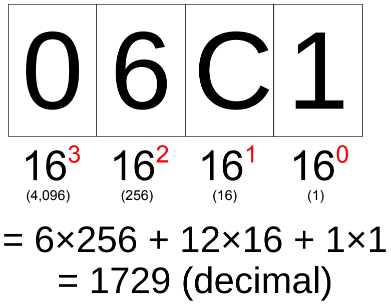
```
00000000.00000000.00000110.11000001
```

We could use spaces to break this into 4-bit nybbles:

```
0000 0000 0000 0000 0000 0110 1100 0001
```

Converting each nybble into its hex equivalent, we'd get:

```
0 0 0 0 0 6 C 1
```



These eight hex digits (`000006C1`) represent the same value as the 32 bits of the number's binary representation. Since hex notation is much more compact than binary, and since it's easy to convert between binary and hex, we often use hexadecimal numbers in computing.

Hex numbers are used so often in computing that the C language has some built-in facilities for dealing with them. For example, we can write hex numbers directly into our program without needing to convert them into decimal. If we wanted to give a variable the value `FF` in hex (which is 255 in decimal), we could say:

```
n = 0xFF;
```

When we start a number with "`0x`", the compiler assumes that the number is written in hexadecimal notation. This might take a little getting used to, since it looks like you're writing "zero times `FF`", but you'll get the hang of it. The zero at the beginning tells the compiler that this is a number, and not a variable name (since variable names can't start with digits), and the `x` means that the number is in he**X**adecimal. (Note that it doesn't matter whether you use an upper-case or lower-case `x`, but programmers usually stick to lower-case.)

Figure 13.52: Symbols like these, often found on the sides of Pennsylvania barns, are called "Hex signs".

Image: Wikimedia Commons

We can also read and write numbers in hex notation. The placeholder "`%x`" means "read or write this number as hex". For example, these statements:

```
n = 1729;
printf ( "In hex the number is %x.\n", n );
```

would print:

```
In hex the number is 6c1.
```

Notice that this printed a lower-case “`c`”. If we wanted to print upper-case letters, we could have used “`%X`” instead, to get `6C1`. If we wanted to print a `0x` at the beginning of the number, as we would if we used the number in a program, we could use “`%#x`”, like this:

```
printf ( "In hex the number is %#x.\n", n );
```

which would print “`In hex the number is 0x6c1.`” As before, using an upper-case `X` would cause the printed letters to be upper-case.

When using “`%x`” with `scanf`, case doesn't matter. For example, these statements:

```
  printf ("Enter number: ");
  scanf("%x",&n);
```

would accept a number written as `6c1` or `6C1`. It would also be OK if you entered `0x6c1` or `0x6C1`. As always, `scanf` tries to be forgiving.

If you've ever created web pages, you've probably already encountered hex numbers. In that context, these numbers are often used to specify colors. For example, if you see “`#FFA500`” that means 100% red, about 65% green, and no blue, which would mix together on your screen to give you a nice orange color. (The color `#0006C1`” is a nice dark blue.)

Figure 13.53: The three pairs of hex digits in this kind of color specification tell the computer how much red, green, and blue to mix together. Each pair of digits is a number between `00` and `FF`.

Color specifications like these consist of three pairs of hex digits, telling the computer how much red, green, and blue to use. Each pair of digits represents one 8-bit byte. The number stored in this byte says how much of that color to use, on a scale from `00` (none of it) to `FF` (all of it). Altogether, the color specification takes up 3 bytes of storage, or 24 bits. You'll often see this referred to as “24-bit color”. This many bits can store any of $2^{24} = 16,777,216$ different values, so that's how many colors it's possible to specify this way.

Let's try writing a little program that uses hex numbers to play with color. Program 13.10 loops through some of the values for R, G, and B and prints the resulting hexadecimal color identifier. Since (as we noted above) there are over 16 million possible R,G,B combinations, the program won't print them all. Instead, it uses only eight out of the possible 256 values for R, G, or B. That means the program will print $8 \times 8 \times 8 = 512$ different colors.

The program has three nested loops. Each loop gives one of the primary colors (R, G, or B) values between `00` and `FF` in steps of 32. That gives eight values for each of the primary colors (since $256/32 = 8$).

Program 13.10: colorcube.cpp

```
#include <stdio.h>
int main () {
  int r,g,b;
  int step=32;

  for ( r=0; r<=0xFF; r+=step ) {
    for ( g=0; g<=0xFF; g+=step ) {
      for ( b=0; b<=0xFF; b+=step ) {
        printf ( "%d %d %d ", r,g,b );
        printf ( "0x" );
        printf ( "%02x", r );
        printf ( "%02x", g );
        printf ( "%02x", b );
        printf ( "\n");
      }
    }
  }
}
```

The program's output has four columns, with each line looking something like this:

```
16 176 48 0x10b030
```

The first three numbers are the R, G, and B values expressed as decimal numbers between 0 and 255. The fourth colum is a single hexadecimal number corresponding to these RGB values.

Note that we've written the number in the format C uses for hex numbers, by starting it with "0x". This makes it easy to use the output with other programs that understand this way of writing hex numbers. After the 0x we write the R, G, and B values as hex numbers. Since we want each of these numbers to have two digits, we can't just use %x as the placeholder. We want to force printf to print two digits, even if the left-hand one is zero. We can make this happen by adding "02" between % and x. The 2 tells printf to always leave room for 2 digits, and the 0 says to put a zero on the left if there would otherwise not be a digit there[11].

Figure 13.54: *Hexe* means "witch" in German. Here's W.W. Denslow's illustration of the Wicked Witch of the West, from L. Frank Baum's *The Wonderful Wizard of Oz*.

Image: Wikimedia Commons

[11] See Appendix E for other printf tricks.

Exercise 65: Color Cube

Create and compile Program 13.10. Run the program and redirect its output into a file, like this:

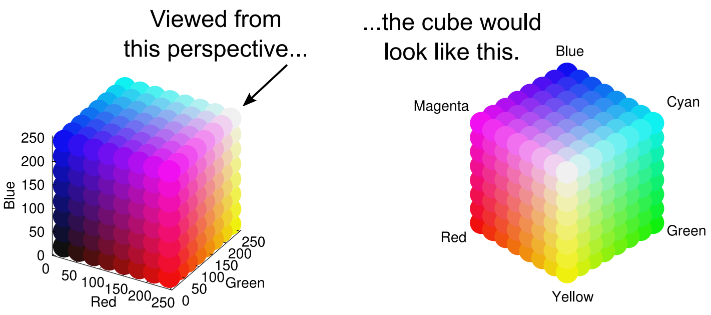
```
./colorcube > colorcube.dat
```

We can use *gnuplot* to visualize the output of our program. We'll use *gnuplot*'s `splot` command to plot points in 3-dimensional space, where the x, y, and z coordinates of each point are the R, G, and B values from our program. The fourth column will determine each point's color. Fortunately, *gnuplot* also uses a leading `0x` to identify hex numbers, just as C does.

Start up *gnuplot* and give it the following commands:

```
set hidden3d
set xyplane 0
set view equal xyz
splot "colorcube.dat" using 1:2:3:4 pt 7 ps 6 lc rgb variable
```

The result should look something like the left-hand figure below:



Try grabbing the cube with your mouse and rotating it around!

How does this *gnuplot* magic work? The first three commands have the following effects:

- `set hidden3d` causes *gnuplot* to "hide" objects that are "behind" others in 3-d plots like this. Without this setting, the stacking of objects depends on the order in which *gnuplot* draws them, which might not have anything to do with which object is "closer" to the viewer.

- `set xyplane 0` causes *gnuplot* to set the cube on the x-y plane. Without this, `splot` will leave some space below the cube.

- `set view equal xyz` causes all of the axes to have equal scales. This makes the plot a cube, rather than a shoebox.

While still in *gnuplot*, try turning each of these options off, one at a time, by using `unset` instead of `set`, and then typing `replot` after unsetting each one. This will show you the effect of each setting.

The last *gnuplot* command (`splot`...) says that we want to use columns 1, 2, 3, and 4 of the file. The "`lc rgb variable`" at the end of the line tells *gnuplot* that the color of each point will be specified in RGB format and will be given by the last column of our data file. The "`pt 7`" and "`ps 6`" control the point type and point size. These choices cause the points to be plotted as large circles.

Figure 13.55: A "hex" can also be a curse. Here's a Roman curse written on a lead tablet. The writing says, in part, *"I curse Tretia Maria and her life and mind and memory and liver and lungs mixed up together..."* Yow! The tablet is in the British Museum.

Image: Wikimedia Commons

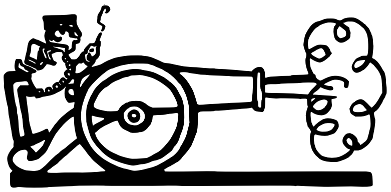
A. Some Challenging Projects

The following pages contain some projects that will challenge you to write programs using the skills you've learned in this book. Give them a try!

Project 1: Cannonball Run

Introduction: The Visitor

Imagine that you're an artilleryman in Napoleon's army. Your job is to fire a cannon, and to drop cannonballs as close as possible to a given target. You take your job seriously, and spend a lot of time thinking about the factors that limit your cannon's accuracy.



Ignoring effects of the wind and rain (which you can't control), you know that if the cannon always fired cannonballs at the same speed and angle, they'd always hit the same spot. But in reality, the speed and angle aren't always the same. Damp gunpowder or badly-formed, ill-fitting cannonballs change the speed, and the cannon doesn't stay in exactly the same position from one shot to the next, tilting a little up or down, or side to side.

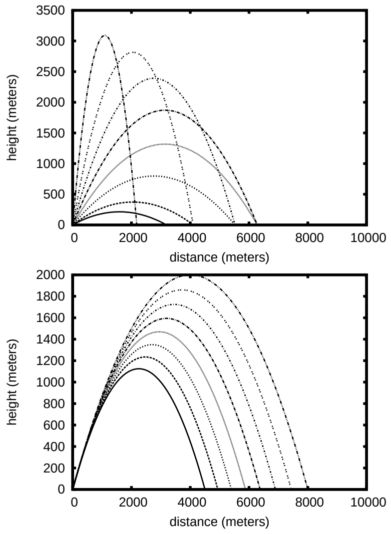


Figure A.1: The figures above illustrate how a cannonball's final position depends on the upward angle at which it's fired (top) and its initial velocity (bottom).

If you could fix even one of these problems you'd deserve a medal! But, sadly, it would take years of experimentation and tons of gunpowder to develop a new cannon design. If only there were some way to accurately simulate a real cannon with something smaller, like the toy cannons that tin soldiers use.

As you're standing beside your cannon, musing about this, a mighty concussion knocks you off your feet! An attack! But no. Rolling onto your stomach and peering through the settling dust you see, not a cannonball's crater, but an oddly-dressed man. He's lying on the ground, waving his hands in the air. "I've done it!", he shouts, "I've done it! I'm the first man to travel back in time!"

Over the course of the next hour you find out that this man has come from the 21st Century, and that the technology of his time is almost magical. The time-traveller has brought with him an object the size of a book which, when unfolded, can display moving images and even play music! The traveller calls it a "computer". This device is the solution to your problem! It can instantly simulate thousands of cannon shots!

Program 1: Simulating the Cannon

This project will require you to write three programs. The first of them will be named "`simulate.cpp`", and it will simulate a cannon. The program will allow the user to specify a speed and vertical angle for the cannonballs, but will add some random "jitter" to these values to simulate the cannon's imperfections. It will also add some random jitter to the side-to-side direction in which the cannon is pointing. The program's output will be a file containing the x, y coordinates at which each simulated cannonball lands[1].

The program should accept all of its parameters on the command line, as described in Section 9.15 of Chapter 9. The usage should be:

```
./simulate nshots vinit theta outfile
```

where:

- `nshots` is the number of cannonballs to fire.
- `vinit` is the ideal initial velocity of the cannonballs (before adding any jitter).
- `theta` is the ideal angle between the cannon and the ground (before adding jitter), expressed in degrees. An angle of zero means the cannon is horizontal, and an angle of 90 means the cannon is pointing straight up into the air. (See Figure A.3.)
- `outfile` is the name of a file into which the program will write the x and y coordinates at which each cannonball lands. Assume that the cannon points along the x axis, but cannonballs may veer by some small random angle, β, to the right or left. (See Figure A.4.)

If the user doesn't supply enough command-line arguments, the program should print out a friendly usage message and then stop without trying to do anything else.

After running the program, the output file should contain two columns of numbers with a space between them. The first column is x and the second column is y.

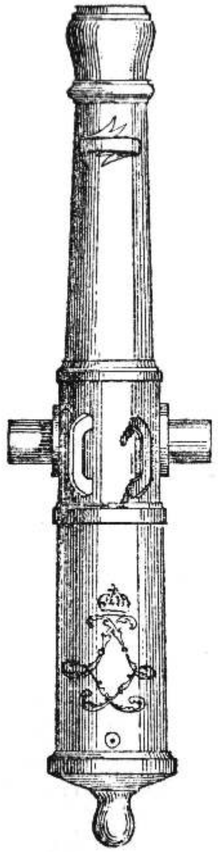


Figure A.2: Canon de 16 Gribeauval.
Source: Wikimedia Commons

[1] Note that in all of the following we'll ignore the effects of air resistance.

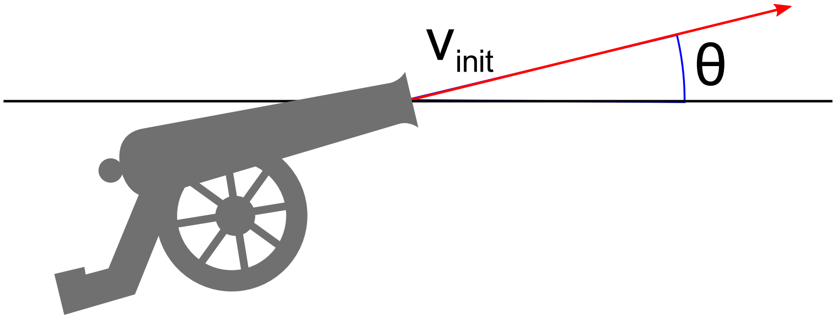


Figure A.3: **Side** view of the cannon's upward angle, θ.

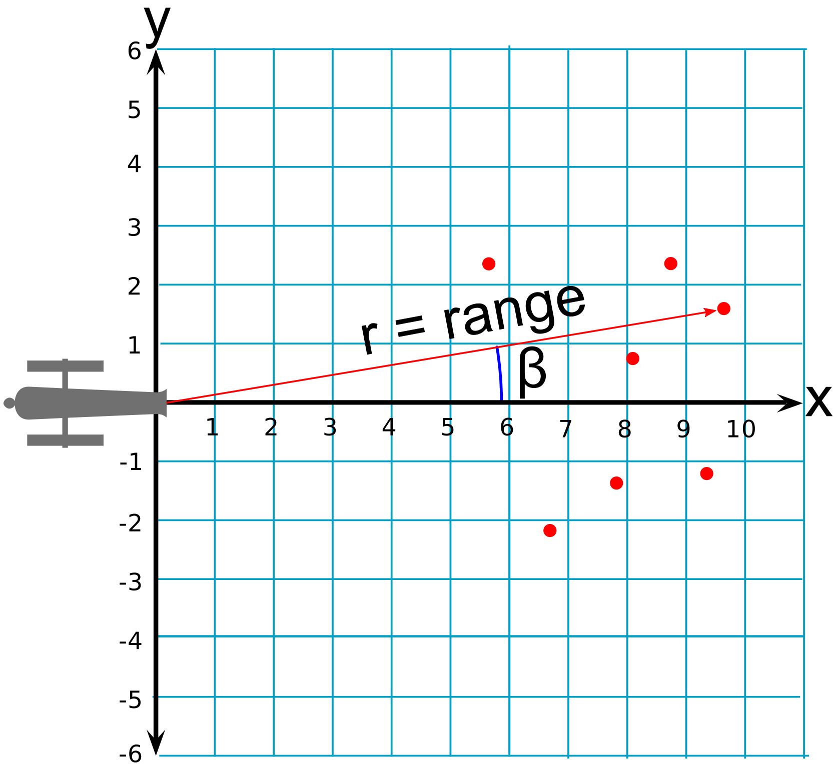


Figure A.4: **Overhead** view of a cannonballs side-to-side angle, β, its range, r, and the landing positions of some cannonballs.

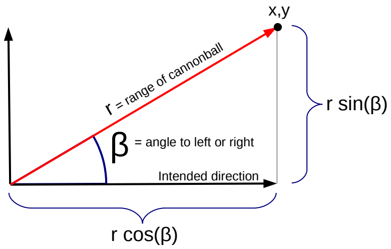


Figure A.5: Finding the x and y coordinates of a cannonball, given its range and the horizontal angle β.

To get you started, the helpful time-traveller has already written much of the program for you (see Program A.1). All you need to do is complete the program by filling in `main` and adding a `help` function that prints out a friendly usage message when the user doesn't supply enough arguments on the command line. As you can see, you'll be using several functions that have appeared in Chapters 9 and 11. These are at the top of Program A.1.

Your program should determine the landing positions of the cannonballs as follows:

1. Convert the upward angle (`theta`) into radians, since C's trigonometry functions use radians instead of degrees. You can use the function `to_radians` to do this. (This function is taken from Chapter 9, Section 9.8.)

2. Open the output file for writing. (See examples like Program 5.3 in Chapter 5.) Note that the name of the file will be in the command-line argument `argv[4]`.

3. Now loop through all of the cannon shots, using a `for` loop.

4. Each time the cannon shoots, set the cannonball's initial velocity and upward angle to the values of `vinit` and `theta`, plus some random "jitter". To do this, use the function named `normal` (taken from Section 11.4 of Chapter 11). The `normal` function generates numbers that tend to be close to zero, but sometimes have other values. (See Figure A.6.)

   - For each shot your program makes, set the cannonball's initial velocity to `vinit + 0.1*vinit*normal()`. This will give a value that tends to be within +/- 10% of the "ideal" velocity, `vinit`.
   - Set each cannonball's upward angle to `theta + 0.01*normal()`. This will give a value that tends to be close the "ideal" angle, `theta`, but has some small random variation.

5. Now that you have the cannonball's velocity and upward angle, use the `range` function (taken from Chapter 9, Section 9.8) to calculate its range. This function takes the cannonball's initial velocity and its upward angle, and returns the cannonball's "range" (the horizontal distance from the launch point to the landing point). (See Figure A.4.)

6. To determine the cannonball's landing position you'll also need to know β, the angle by which its path deviates to the right or left. (See Figure A.4.) Use the `normal` function for this by setting β equal to `0.01*normal()`. This will give you a random, small angle.

7. Now that you have the cannonball's range and the angle β, you calculate the x and y coordinates of its landing spot. See Figure A.5.

8. Finally, write the x and y coordinates into the output file. (See examples like Program 5.3 in Chapter 5 if you don't remember how to do this.)

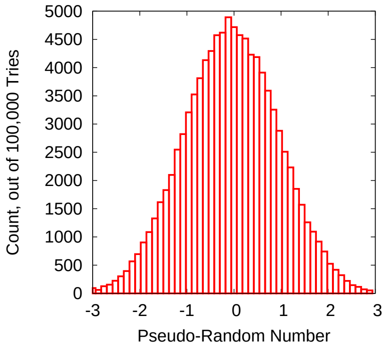


Figure A.6: The `normal` function generates pseudo-random numbers that are most likely to be near zero, with smaller probabilities for other values. This figure shows 100,000 pseudo-random numbers generated by `normal`.

Once you've written and compiled your program, run it like this to produce an output file to use with your next program:

```
./simulate 10000 250 45 simulate.dat
```

Program A.1: simulate.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }

  return ( rand()/(1.0+RAND_MAX) );
}

double normal () {
  int nroll = 12;
  double sum = 0;
  int i;

  for ( i=0; i<nroll; i++ ) {
    sum += rand01();
  }

  return ( sum - 6.0 );
}

double g = 9.81; // Acceleration of gravity.

double to_radians ( double degrees ) {
  return ( 2.0 * M_PI * degrees / 360.0 );
}

double time_of_flight ( double v0, double angle ) {
  double t;
  double vy0;
  vy0 = v0 * sin(angle);
  t = 2.0 * vy0 / g;
  return ( t );
}

double range ( double v0, double angle ) {
  double d;
  d = v0 * cos(angle) * time_of_flight( v0, angle );
  return ( d );
}

int main (int argc, char *argv[]) {

        //
        // Insert program here!
        //
}
```

Program 2: Analyzing the Results

Your second program will be named `analyze.cpp`. It will read a data file produced by your first program, and give you a statistical summary of the data it contains.

Like the first program, `analyze` should accept all of its parameters on the command line, and give users a helpful message if they don't give it the right number of arguments. The usage should be:

```
./analyze filename
```

where `filename` is the name of a data file produced by your `simulate.cpp` program.

The output of the `analyze` program should look like this:

```
Average x = 6428.287617
Std. dev. of x = 1286.944844
Min x = 2568.660526
Max x = 13046.427659
Average y = -0.611109
Std. dev. of y = 65.978704
Min y = -284.001774
Max y = 313.589122
```

showing the average values of x and y, the standard deviations of x and y, and the minium and maximum values of x and y.

The helpful time-traveller has come to your aid again, and written some of the program for you (see Program A.2). You'll just need to fill in `main` and write a `help` function.

To analyze the data, the program should proceed as follows:

1. First, open the data file for reading. See Program 7.5 in Chapter 7 for an example of this. Refer to that program to see how to read the data and calculate the average and standard deviation.
2. The time-traveller has kindly provided you with an easy way to find minimum and maximum values, using the two functions `findmin` and `findmax`. Each time you read a new value of x, for example, just say `xmax = findmax(x,xmax,n)`. This will update the value of `xmax` if necessary. When you're done reading all of the data, `xmax` will contain the largest value of x.

Run your program to analyze the `simulate.dat` file you produced earlier. Check to make sure its results look realistic. (Compare them to the sample output above.)

Program A.2: analyze.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
double findmax ( double x, double oldvalue, int n ) {
  if ( n == 0) {
    oldvalue = x;
  } else {
    if ( x > oldvalue ) {
      oldvalue = x;
    }
  }
  return ( oldvalue );
}
double findmin ( double x, double oldvalue, int n ) {
  if ( n == 0 ) {
    oldvalue = x;
  } else {
    if ( x < oldvalue ) {
      oldvalue = x;
    }
  }
  return ( oldvalue );
}
int main ( int argc, char *argv[] ) {

        //
        // Insert program here!
        //

}
```

Program 3: Making Pictures

Your final program will be called `visualize.cpp` and it will let you make pictures like the ones shown in Figure A.8. These figures show the distribution of landing positions of 10,000 simulated cannonballs.

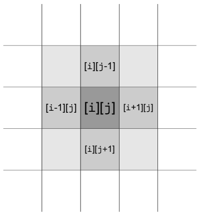


Figure A.7: Each element of `grid` records the number of cannonballs that landed within a particular section of the battlefield.

The figures represent 2-dimensional histograms. We talked about histograms in Chapter 7, but we didn't say much about 2-dimensional ones. Because of that, our friendly time-traveller has written almost all of this program for you. (See Program A.3.)

This program uses a 2-dimensional, `nbins` × `nbins` array named `grid`. Each element of the array represents an area of the battlefield. The number stored in each element is the number of cannonballs that landed in that area.

Like the preceding programs, this one will expect parameters on its command line. Its usage will be:

```
./visualize xmin xmax ymin ymax infile outfile
```

where `xmin`, `xmax`, `ymin`, and `ymax` specify the limits of rectangular area of the battlefield. `infile` is the name of a data file produced by your `simulate` program. `outfile` is the name of a file into which the current program will write its results.

Two key parts of the program have been left for you to fill in. First, near the top of `main`, you need to set all of the elements of `grid` to zero. To do this, you'll need two nested "`for`" loops. Inside the loops, set each element, `grid[xbin][ybin]`, to zero.

Second, near the end of `main`, you need to open the output file for writing and write your results into it. (You'll again need two nested "`for`" loops to do this.)

The file should have three columns, `x`, `y`, and `grid[xbin][ybin]`, where `x` and `y` are the coordinates of the center of the grid element. Use `x=xmin+xbinwidth*(0.5+xbin)`, and `y` similarly, for the center position of each grid element.

There should also be a blank line after every `nbins` rows. See the end of Section 6.12 for an explanation of this blank line, and the last part of Program 6.8 for an example showing how to create it.

After writing and compiling the program, try it out. Use your `analyze` program to find good values for `xmin`, `xmax`, `ymin`, and `ymax`. Use these values and your newest program to process the data in `simulate.dat` and create a new file, `visualize.dat`, that can be plotted with *gnuplot*:

```
./visualize 2569 13046 -284 314 simulate.dat visualize.dat
```

Try plotting your results with *gnuplot*. To produce the top graph in Figure A.8, give *gnuplot* the following command:

```
plot "visualize.dat" with image
```

To produce the bottom graph in Figure A.8, use this *gnuplot* command:

```
splot "visualize.dat" with image, "" with histeps
```

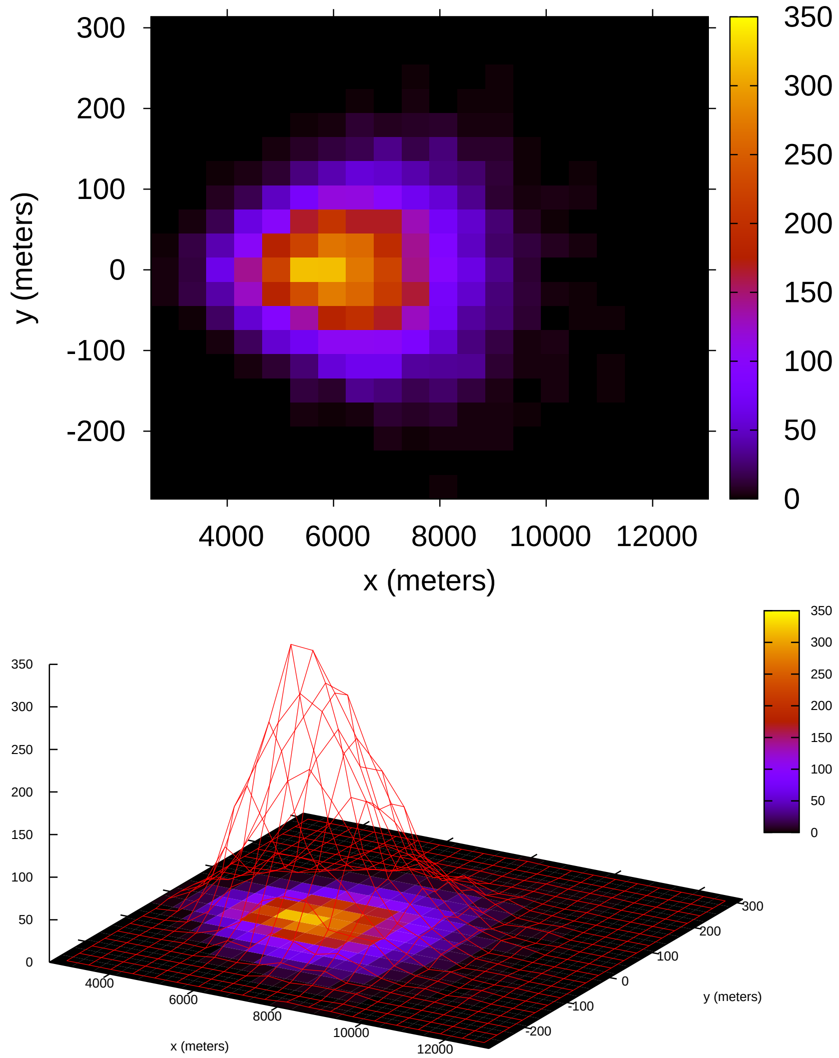


Figure A.8: Two views of the distribution of cannonball landing positions. The color scale shows how many cannonballs (out of 10,000) landed in each grid element.

Program A.3: visualize.cpp

```
#include <stdio.h>
#include <stdlib.h>

void help() {
  printf ("Usage: ./visualize xmin xmax ymin ymax input.dat output.dat\n");
}

int main ( int argc, char *argv[] ) {
  const int nbins = 20;
  int grid[nbins][nbins];
  double x, y;
  double xmin, xmax;
  double ymin, ymax;
  double xbinwidth, ybinwidth;
  FILE *output;
  FILE *input;
  int xbin, ybin;

  if ( argc != 7 ) {
    help();
    exit(1);
  }

  // Insert code here to reset all bins to zero.

  xmin = atof(argv[1]);
  xmax = atof(argv[2]);
  ymin = atof(argv[3]);
  ymax = atof(argv[4]);

  xbinwidth = (xmax - xmin)/(double)nbins;
  ybinwidth = (ymax - ymin)/(double)nbins;

  input = fopen ( argv[5],"r" );

  while ( fscanf(input, "%lf %lf", &x, &y) != EOF ) {
    xbin = (x-xmin)/xbinwidth;
    ybin = (y-ymin)/ybinwidth;
    if ( xbin >= 0 && ybin >= 0 && xbin < nbins && ybin < nbins ) {
      grid[xbin][ybin]++;
    }
  }

  fclose ( input );

  // Insert code here to open the output file and write
  // the contents of "grid" into it.
}
```

Last Words

As your friend from the future fades away in a cloud of sparkles, you stand there savoring your brief glimpse of the future. "If only we had such technology today," you sigh, as you hear your commander shout the order to begin breaking camp.

Figure A.9: Wellington at Waterloo.

Source: Wikimedia Commons

While you prepare to march into Russia during the Spring of 1812, far away in England a mathematician named Charles Babbage is looking at mathematical tables, like the ones used by artillerymen for aiming their cannons, and thinking about how these tables could be generated automatically, by machinery instead of humans.

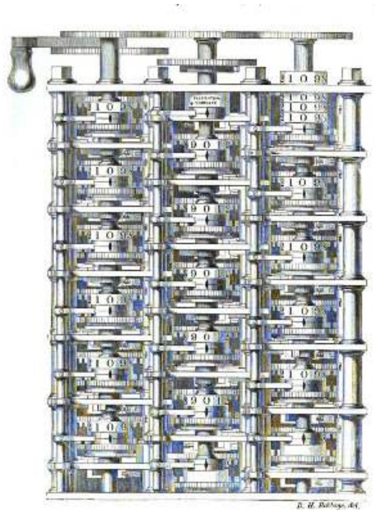


Figure A.10: Part of Babbage's "Difference Engine".

Source: Wikimedia Commons

After Napoleon's defeat at Waterloo in 1815, Babbage exchanges ideas with other mathematicians, English and French, and in 1822 he begins work on the series of computing machines that will become the ancestors of all modern computers.

Figure A.11: The Emperor Napoleon (left), and Babbage's brain (right).

Source: Wikimedia Commons 1, 2

Project 2: Diffusion Confusion

Introduction: Randomly-Bouncing Molecules

Imagine that you're in a large room full of perfectly still air. At the opposite end of the room is a just-opened bottle of perfume. The volatile molecules from the perfume have started to wander out into the room, bouncing off of molecules in the air. How long would it take these molecules to bounce their way across the room to your nose?

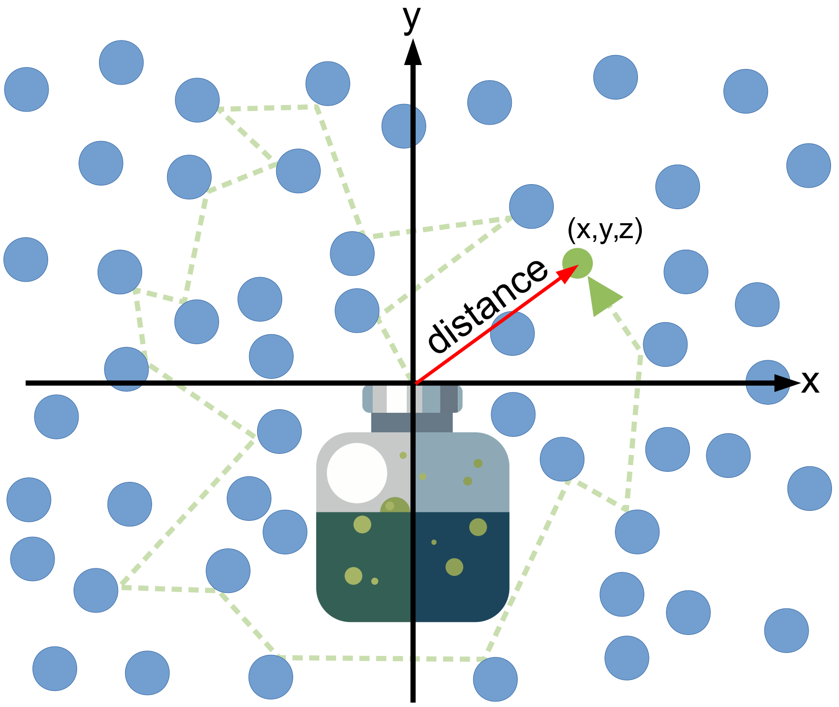


Figure A.12: A molecule leaves the perfume bottle, then bounces around among the air molecules for a while, ending up at a position (x, y, z) some distance from where it started.

A typical speed for a molecule in air is about 1,000 miles per hour, but our perfume molecules don't travel in a straight line. Figure A.12 shows a typical perfume molecule's path. Since it bounces around at random, it tends to linger near the bottle for a long time. The process by which molecules spread out by bouncing around this way is called "diffusion".

In this project you will write three programs: `simulate.cpp`, `analyze.cpp`, and `visualize.cpp`. The first will simulate the paths of perfume molecules through air, the second will analyze the simulated data, and the third will help us visualize one of the results.

Program 1: Simulating the Paths of Molecules

Your first program will be named `simulate.cpp`. It will track the random movement of some number of perfume molecules as they undergo some number of collisions. The program will write the final position of each molecule, and how long it took the molecule to get there, into an output file.

The perfume molecule's path is an example of a random walk, and this program will be very similar to Practice Problem 5 in Chapter 7. One difference is that the new program tracks a random path in three dimensions instead of two, so you'll need to keep track of the molecule's x, y, and z coordinates. Another difference is that we won't assume that each step of the path has the same length, as we did in the earlier program. This time, we'll let the step length vary a little. Each step in the molecule's random path will be the distance from one collision to the next. Finally, the new program won't bother with keeping track of sums or averages.

The program should accept all of its parameters on the command line, as described in Section 9.15 of Chapter 9. The usage should be:

```
./simulate nparticles ncollisions output.dat
```

where:

- `nparticles` is the number of perfume molecules we want to simulate.
- `ncollisions` is the number of collisions each molecule will experience.
- `output.dat` is the name of a file into which the program's results will be written.

If the user doesn't supply enough command-line arguments, the program should print out a friendly usage message and then stop without trying to do anything else. See Section 9.15 of Chapter 9 for an example showing how to do this.

After running the program, its output file should contain four columns of numbers: The x,y, and z coordinates where the molecule ended up, and the time it took to get there. We'll measure time in microseconds (1 microsecond = 10^{-6} seconds) and distances in microns (1 micron = 10^{-6} meters).

Each time a perfume molecule collides with an air molecule, we'll need to generate a new random direction for it, and a new random distance to the next collision. In 3-dimensional space, we can describe a particle's direction with two angles, θ (theta) and φ (phi) (see Figure A.13):

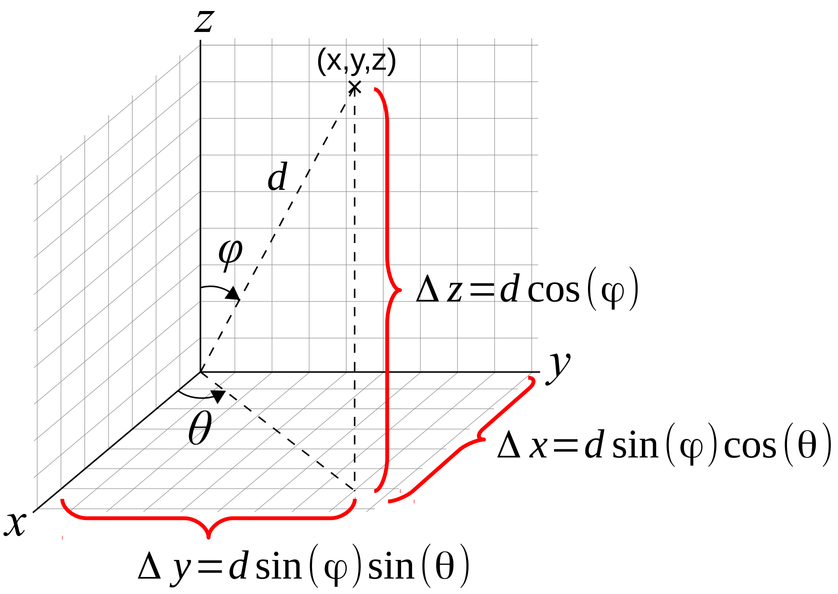


Figure A.13: After a collision, the molecule's new direction is given by two angles, θ and φ. The distance to the next collision is d.

- The angle θ can point in any direction away from the Z axis. It can have any value between zero and 2π radians (360°).
- The angle φ can have any value between straight up (zero) and straight down (π radians, or 180°).

The distance, d, will vary around some average value called the "mean free path", which we'll assume to be 0.14 microns. Each time we generate a value for d we'll do so by adding a little bit of random "jitter" to this distance.

To get you started, I've already written some of the program for you (see Program A.4). All you need to do is complete the program by filling in `main`. As you can see, you'll be using two functions that have appeared in Chapter 11. These are at the top of Program A.4. You'll also see that I've defined the values of the mean free path (`meanpath`) and the speed of the molecules (`speed`), which we assume to be 500 microns/microsecond.

Figure A.14: Trading card for Hoyt's German Cologne, circa 1900.

Source: *Wikimedia Commons*

To track the molecules, your program should do the following:

1. Open the output file for writing[2]. The output file name will be given by `argv[3]`, so you can say something like "`output = fopen(argv[3],"w");`".

2. You'll need a pair of nested `for` loops: An outer loop for each molecule, and an inner one for each collision[3].

3. Keep track of the molecule's position with three variables, `xpos`, `ypos`, and `zpos`. Keep track of the time elapsed with a variable named `t`. Remember to set all of these back to zero whenever you begin tracking a new molecule.

4. Every time the molecule collides, do the following:

   (a) Generate two random angles like this:

   ```
   theta = 2.0*M_PI*rand01();
   phi = M_PI*rand01();
   ```

   (b) Generate a random distance like this:

   ```
   d = meanpath * ( 1.0 + 0.1*normal() );
   ```

   where `normal` is a function shown in Program A.4 below.

   (c) Add Δx, Δy, and Δz (as shown in Figure A.13) to the values of `xpos`, `ypos`, and `zpos`, respectively, to get the molecule's new position[4].

   (d) Update `t` by adding `d/speed` to it. This is the time it will take the molecule to travel the distance `d`.

5. Use the trick described in Section 4.4 of Chapter 4 to print out progress reports as your program is running. After every 10 molecules, print a message like this on the screen: `Working on molecule 10...` (or 20, or 30, and so on). It's OK if the program prints "`Working on molecule 0`" when it starts.

[2] For a reminder about how to write output into files, see examples like Program 5.3 in Chapter 5.

[3] This is similar to what we did in Program 2.7 in Chapter 2.

[4] If you're not familiar with the symbols in Figure A.13, remember that θ is `theta` and φ is `phi`. These are the random angles you generated in step (a) above.

6. After tracking the molecule through `ncollisions` collisions, write `xpos`, `ypos`, `zpos`, and `t` into the program's output file[5]. These should be written as four numbers separated by single spaces, with a `\n` at the end of the line.

Once you've written and compiled your program, run it like this to produce an output file to use with your next program:

```
./simulate 1000 16000 simulate-16000.dat
```

This should produce an output file (`simulate-16000.dat`) containing the final positions and times for 1,000 perfume molecules after each of them bounces 16,000 times.

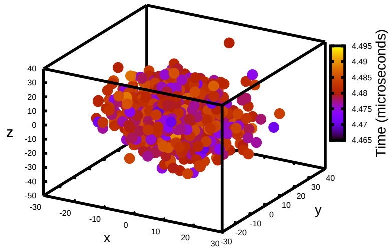


Figure A.15: You can check your first program's results by plotting them with *gnuplot*. This figure shows what you should see if you type:
`splot "simulate-16000.dat" with points palette pointsize 3 pointtype 7`
It shows the final x, y, and z positions of the molecules, color-coded by how long it took them to get there.

Program A.4: simulate.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }
  return ( rand()/(1.0+RAND_MAX) );
}

double normal () {
  int nroll = 12;
  double sum = 0;
  int i;

  for ( i=0; i<nroll; i++ ) {
    sum += rand01();
  }
  return ( sum - 6.0 );
}

int main ( int argc, char *argv[] ) {

  double meanpath = 0.14; // Microns per collision
  double speed = 500; // Microns per microsecond

        //
        // Insert program here!
        //
}
```

[5] See examples like Program 5.3 in Chapter 5.

Program 2: Analyzing the Results

Your second program will be named `analyze.cpp`. It will read a data file produced by your first program, and give you a statistical summary of the data it contains.

Like the first program, `analyze` should accept all of its parameters on the command line, and give users a helpful message if they don't give it the right number of arguments. The usage should be:

```
./analyze input.dat
```

where `input.dat` is the name of a data file produced by your `simulate.cpp` program.

The output of the `analyze` program should look like this:

```
Average distance = 16.292850 microns
Std. dev. of distance = 6.987062 microns
Min distance = 0.684207 microns
Max distance = 45.581858 microns
Average time = 4.480129 microseconds
Std. dev. of time = 0.003552 microseconds
Min time = 4.469744 microseconds
Max time = 4.491567 microseconds
Diffusion Coefficient is 0.29626 cm^2/s
```

where `distance` is the final distance of a molecule from the origin, which is given by

$$distance = \sqrt{x^2 + y^2 + z^2}$$

and `time` is the amount of time the molecule took to get there, which is just the fourth column in your data file.

The "Diffusion Coefficient" is a way of measuring how fast molecules diffuse through the air. It's usually given in units of cm^2/s. If your program calls the average distance `davg` and the average time `tavg`, you can calculate the diffusion coefficient like this:

```
dcm = davg/1.0e4;
tseconds = tavg/1.0e6;
dcoeff = dcm*dcm/2.0/tseconds;
```

where `dcm` is the distance converted to centimeters and `tseconds` is the time converted to seconds. `dcoeff` is the Diffusion Coefficient. It should end up having a value of around 0.3 cm^2/s if your programs are working properly.

Figure A.16: Broken glass perfume amphora from Ephesus, 2^{nd} century CE.

Source: Wikimedia Commons

Again, I've already written some of the program for you (see Program A.5). You'll just need to fill in `main`.

To analyze the data, the program should proceed as follows:

1. First, open the data file for reading. See Program 7.5 in Chapter 7 for an example of this. Refer to that program to see how to read the data and calculate the average and standard deviation.

2. At the top of Program A.5 below I've provided you with an easy way to find minimum and maximum values, using the two functions `findmin` and `findmax`. Each time you read a new value of t, for example, just say `tmax = findmax(t,tmax,n)`, where `n` is the number of molecules you've processed so far. This will update the value of `tmax` if necessary. When you're done reading all of the data, `tmax` will contain the largest value of t. **Note:** It's important that `n` be equal to zero the first time you use these functions.

3. After reading all of the data from the input file, calculate the Diffusion Coefficient (as shown above) and print all of the results.

Figure A.17: "The Perfume Maker", by Rudolf Ernst.

Source: Wikimedia Commons

Program A.5: analyze.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
double findmax ( double x, double oldvalue, int n ) {
  if ( n == 0) {
    oldvalue = x;
  } else {
    if ( x > oldvalue ) {
      oldvalue = x;
    }
  }
  return ( oldvalue );
}
double findmin ( double x, double oldvalue, int n ) {
  if ( n == 0 ) {
    oldvalue = x;
  } else {
    if ( x < oldvalue ) {
      oldvalue = x;
    }
  }
  return ( oldvalue );
}
int main ( int argc, char *argv[] ) {

        //
        // Insert program here!
        //

}
```

Program 3: Visualizing the Distance

Your final program will be called `visualize.cpp` and it will let you make pictures like the one shown in Figure A.18. This figure shows the distribution of final distances of 1,000 perfume molecules after 16,000 collisions.

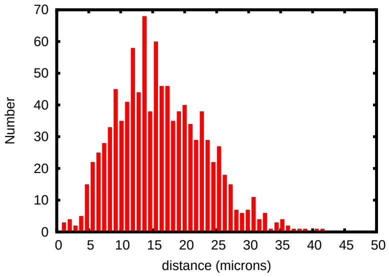


Figure A.18: Distribution of the final positions of 1,000 perfume molecules after each has experienced 16,000 collisions.

This figure is a histogram, like the ones we discussed in Chapter 7. Your third program will be similar to Program 7.1 in that chapter. Again, to get you started, I've written part of the program for you (see Program A.6 below). Notice that I've defined a 50-element array, `bin`, to hold the histogram data.

Like the preceding programs, this one will expect parameters on its command line, and should complain and exit if it doesn't get the proper number of parameters. Its usage will be:

```
./visualize dmin dmax input.dat output.dat
```

where `dmin` and `dmax` are the minimum and maximum distances (as determined by your `analyze` program) `input.dat` is the name of a file produced by your `simulate` program, and `output.dat` is a file into which your new program will write the histogram data.

The output file should contain two columns of numbers, separated by a single space. Unlike Program 7.1, the first column here will contain a distance instead of a bin number (see below for instructions about converting bin number to distance). The second column will be the number of molecules in that bin.

To make the histogram, the program should proceed as follows:

1. First, determine the `binwidth`, like this:

   ```
   binwidth = (dmax-dmin)/nbins;
   ```

2. Next, use a `while` loop to read data from the input file. Each line of the file will contain four values: x, y, z, and t.

3. Every time you read a line, determine the distance from $distance = \sqrt{x^2 + y^2 + z^2}$.

4. Determine which bin this distance belongs in, and increment that bin. Be sure to keep a count of the number of over/underflows, as Program 7.1 does.

5. After processing all of the input data, write the histogram data into

the output file. For each bin of the histogram, write two numbers separated by a single space: the distance represented by that bin, and the number of molecules that fell within it. The distance can be calculated from the bin number, like this:

```
distance = dmin + binwidth*(0.5+i);
```

where `i` is the bin number.

6. Finally, at the bottom of the output file, write a line beginning with a # that tells how many overflows or underflows were seen.

Run your program like this to make a histogram of the data you produced earlier:

```
./visualize  0.684207 45.581858 simulate-16000.dat visualize-16000.dat
```

You can plot the resulting data file with *gnuplot* like this:

```
plot "visualize-16000.dat" with impulses lw 5
```

The result should look like Figure A.18.

Program A.6: visualize.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main ( int argc, char *argv[] ) {

  const int nbins=50;
  int bin[nbins];

        //
        // Insert program here!
        //

}
```

Results

What do your results tell you? If you were to run your `simulate` program two more times, like this:

```
./simulate 1000 1000 simulate-1000.dat
./simulate 1000 4000 simulate-4000.dat
```

and then use your `analyze` program to analyze each of these files and your `simulate-1600.dat` file, you might notice a pattern. Every time you increase the number of collisions by a factor of four, the average distance increases by a factor of two. This fact is reflected in the definition of the Diffusion Coefficient, which tells us that the time it takes molecules to travel a given distance by diffusion is:

$$t = \frac{d^2}{2D}$$

where t is the time, d is the distance, and D is the diffusion coefficient.

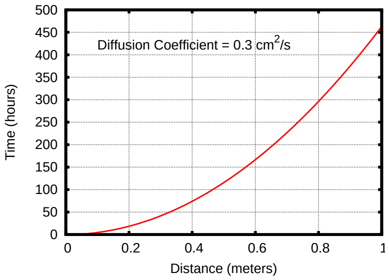


Figure A.19: How long would it take our perfume molecules to diffuse across a room? A long time!

If we plotted time versus distance, we'd get a graph like Figure A.19. As you can see from the graph, it would take hundreds of hours for our perfume molecules to travel even one meter. Diffusion is apparently very slow! Scents usually reach our nose by riding on air currents, rather than through diffusion.

Why is diffusion so slow? From Chemistry class we know that a small amount of air (say, a ballon full) contains on the order of 10^{23} molecules. That's a lot of obstacles to bounce off of. Even though our perfume molecule might be traveling at 1,000 miles per hour, it collides with air molecules billions of times per second, and each collision sends it off in another random direction.

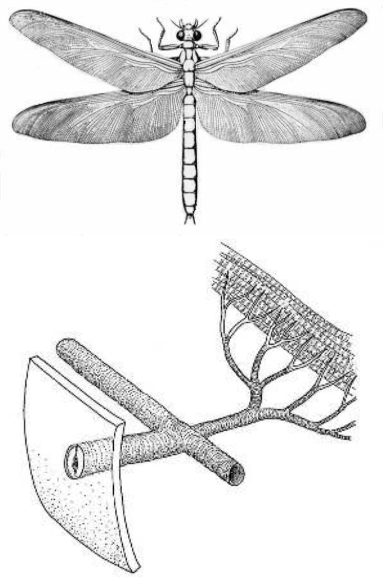


Figure A.20: In the Carboniferous period Earth's oxygen levels were much higher than they are today. This allowed giant inects like the dragonfly *Meganeura* (top) to survive, even without lungs. *Meganeura* had a two-foot wingspan! The bottom illustration shows tracheae inside an insect's body.

Source: *Wikimedia Commons* and *D.G. Mackean*

The low speed of diffusion explains why we have lungs, and why there aren't any human-sized insects. Breathing moves oxygen by two mechanisms: *diffusion* and *advection*. When we breath, air is drawn into our lungs by advection (the bulk motion of a fluid) and it brings oxygen molecules along with it. When the air gets down into our lungs, oxygen molecules then diffuse through the thin walls of blood vessels. This is a very short distance, so diffusion can do the job relatively quickly. The blood then carries the oxygen all through our body (advection again).

Insects don't have lungs. Their bodies contain hollow tubes called *tracheae* that open to the outside world. Oxygen molecules wander into these tubes by diffusion, and then wander through the tubes until they reach cells inside the insect's body. This is a slow process, but since insects are small, the distances are short. If insects were human-sized, they couldn't get oxygen quickly enough through this mechanism.

Project 3: Proton Power

Introduction: Particle Beam Therapy

We all know that radiation can cause cancer, but radiation can also be used to fight cancer. One example of this is particle beam cancer therapy, in which a beam of charged particles (usually protons or pions) is shot into a tumor with the goal of destroying it.

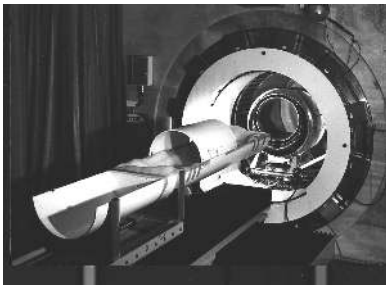


Figure A.21: An apparatus used for pion-beam radiation therapy at the Paul Scherrer Institut. The patient lies in the semicircular cradle, which is inserted into the apparatus behind during treatment.

As particles from such a beam travel through the body, they gradually lose energy and eventually come to rest. As it turns out, much of the particle's energy is lost close to the point at which it stops. This makes such beams well-suited for killing tumors without doing too much damage to the other tissues they pass through on the way to the tumor, or tissues beyond the tumor.

Particles with higher energies will travel farther into the body. By adjusting the energy of the particles, we can cause them to stop at a chosen depth (ideally, inside a tumor).

At moderate energies, a beam of particles traveling through a body loses energy mostly through interactions with electrons. Although it's possible that some of the particles will bump into an atomic nucleus, that doesn't happen very often. Since protons are 2,000 times heavier than electrons, beams of these particles tend to travel in a straight line, knocking puny electrons aside as they go.

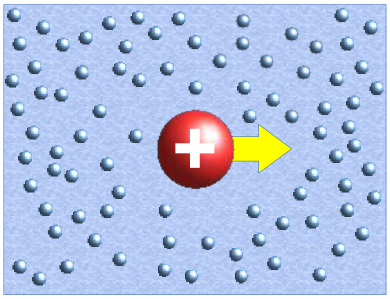


Figure A.22: A proton (shown with a plus sign because of its positive charge) is much larger than the electrons it knocks aside while travelling through the body.

Figure A.23 shows how much energy protons deposit as they travel through the body. The four curves show what happens when you use protons of four different starting energies, ranging from 50 MeV to 125 MeV. The energy deposited damages the body's tissues. The goal is to destroy the tumor without doing too much damage to healthy tissue.

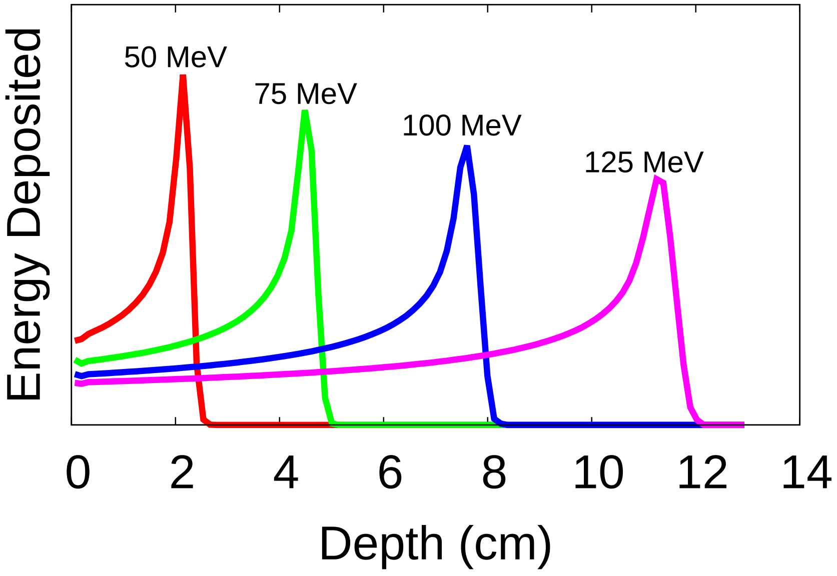


Figure A.23: Energy deposited at various depths by incoming protons having energies of 50, 75, 100, or 125 MeV. As you can see, more energetic protons penetrate to greater depths. Also notice that most of a proton's energy is deposited near its stopping point.

The Assignment

Imagine you're a doctor working at a radiation therapy facility. You have at your command a beam of protons. You can aim the beam precisely, and control its energy.

You're preparing for a visit by a patient with a 2-centimeter-thick tumor buried 8 centimeters deep in her body (see Figure A.24). You need to determine what energy the protons should have in order to deposit most of their energy in the region of the tumor.

A physicist colleague has given you a formula to calculate the energy lost by a particle while going through a thin slice of material. The formula has a form like this[6]:

$$\Delta E = \Delta x \cdot f(E, proton\ properties, material\ properties)$$

where ΔE is the amount of energy the particle loses, Δx is the thickness of the slice, and f is some function that depends on E (the energy at the beginning of the slice) as well as the constant properties of the particle (like charge and mass) and properties of the material (like density).

Unfortunately, your physicist friend tells you that eight centimeters is too big to call a "thin slice". But that's OK, she says. Just treat the eight centimeters as though it was a stack of thinner slices, as shown in Figure A.25. Each time the proton passes through one of the slices,

[6] The actual equation is called the Bethe-Bloch formula.

it loses some amount of energy, ΔE. This lost energy damages the tissue in that slice. The proton then enters the next slice with its energy reduced by the amount ΔE.

Your assignment is to write three programs: **`simulate.cpp`**, **`visualize.cpp`**, and **`analyze.cpp`**. The first will simulate the passage of protons through the patient's body, the second will help visualize these results, and the third will help choose the right proton energy.

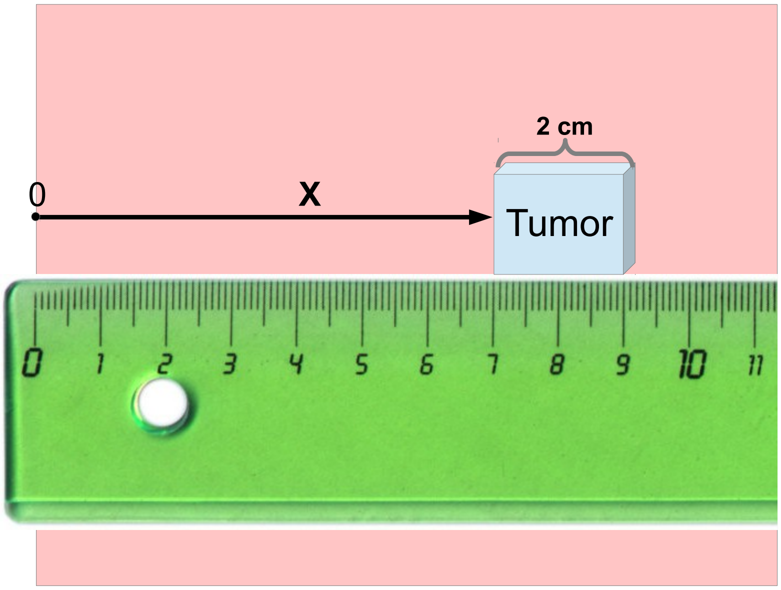


Figure A.24: Our patient's tumor is 2 cm thick, and is centered at a depth of 8 cm. Here "x" represents the depth below the patient's skin.

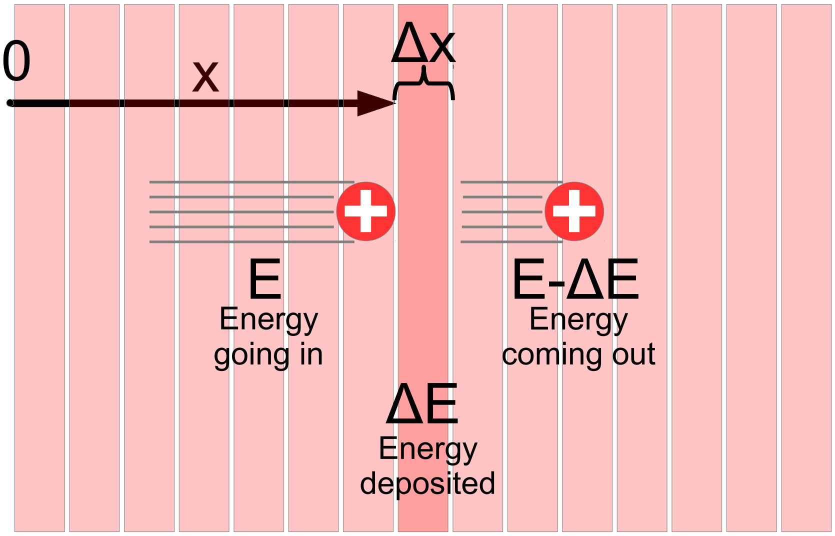


Figure A.25: We can look at the patient's body as a series of thin slices through which the proton must pass. Each time the proton passes through one of the slices, it loses some amount of energy, ΔE. This lost energy damages the tissue in that slice.

Program 1: Simulating Protons

Your first job will be to write a program named **simulate.cpp** that keeps track of the energy that protons lose as they travel through such a stack of thin slices. Each slice will have a thickness of 0.01 cm. Assume each proton travels in a straight line, starting at `x = 0` and progresses along the `x` axis until it runs out of energy. Each time a proton passes through a slice, the program should write the proton's position, energy loss, and remaining energy into an output file.

Your physicist friend has kindly provided you with the beginning of a program, but she's too busy to finish it. The part she's written for you is shown in Program A.7. Near the top of the program are some numbers you'll need. The program assumes that humans are just made out of water, since they mostly are.

She's also written some useful functions in a header file named **dedx.h**, which is shown below as Program A.8. The biggest function in it is named `dEdx`, and it does most of the work of calculating how much energy a proton loses while going through one of the slices. Notice that `simulate.cpp` has an `include` statement near the top that fetches `dedx.h`.

Figure A.26: The international symbol for ionizing radiation, which was first used at Berkeley Radiation Laboratory in 1946.

Source: *Wikimedia Commons*

Program A.7: simulate.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "dedx.h"

int main (int argc, char *argv[]) {
  double pmass = 938.27;     // MeV, Proton mass.
  double pcharge = 1.0;      // Proton charge.
  double rho = 1.0;          // Density, g/cm^2, for water.
  double amass = 18.01;      // Atomic mass, AMU, for water.
  double anum = 10.0;        // Atomic number, Z, for water.
  double activation = 75.0; // Activation energy, eV, for water.

  double dx = 0.01;          // Slice thickness, cm.

  int nprotons;
  double estart, energy;
  double x, de;
  FILE *output;

  // Sorry! got to run to a faculty meeting. You'll
  // have to insert the rest of the program here.

}
```

To complete the program, you'll need to do the following:

1. First, copy Program A.8 (`dedx.h`) into a file named `dedx.h` and save it. Then create a file named `simulate.cpp` and start by putting the contents of Program A.7 into it. This will be the program that does your proton simulation.
2. Your program should accept three arguments on the command line.[7] When you're done writing your program, you should be able to run it like this:

   ```
   ./simulate nprotons estart output
   ```

   where:

   - `nprotons` is the number of protons you want to simulate.
   - `estart` is the starting energy of the protons.
   - `output` is the name of an output file into which the program will write its results.

   If the user doesn't supply enough command-line arguments, the program should print out a friendly usage message and then stop without trying to do anything else[8].

   Since `nprotons` is an integer, you'll need to use the `atoi` function to convert this command-line argument[9]. For `estart` you'll need to use `atof`, since this number can contain decimal places. The output file name won't need any conversion, since it's already a character string. You can just use that argument directly, like this:

   ```
           output = fopen( argv[3], "w" );
   ```

3. Your program will need a pair of nested loops: An outer "`for`" loop that generates protons, one a at a time, and an inner "`do-while`" loop that tracks each proton through the slices until the proton loses all of its energy.

4. Each time the program starts tracking a new proton it should set the proton's initial position and energy. To be more realistic, the program should add some "wiggle" to these values. In the real world, the particles in a proton beam don't all have exactly the same energy, and they won't necessarily enter the body at exactly the same spot (the patient might move a little, for example). Use the function named "`normal`" (defined in `dedx.h`) for this. Here's how to do it:

   ```
   energy = estart + 0.01*estart*normal();
   ```

Figure A.27: A "wind" of charged particles, including many protons, blows outward from the Sun. It interacts with the earth's magnetic field to produce the aurora.
Source: Wikimedia Commons

[7] We learned how to use command-line arguments in Sections 9.15 and 9.16 of Chapter 9.

[8] See Section 9.16 of Chapter 9 for information about how to do this.

[9] See Problem 7 (`add.cpp`) at the end of Chapter 9.

```
x = 0 + 0.1*normal();
```

This sets the proton's initial energy to `estart` ± 1% and the starting position to zero cm ± 1 mm.

5. Each time a proton goes through a slice of tissue, your program should do the following:

   (a) Calculate the amount of energy the proton deposits in the slice (we'll call that "`de`"). Our physicist friend has given us the function named `dEdx` to help us calculate this.

   ```
   de = dx * dEdx(energy, pmass, pcharge, rho, amass, anum, activation);
   ```

   (b) Calculate the proton's new energy:

   ```
   energy = energy - de;
   ```

   (c) Update the proton's position:

   ```
   x = x + dx;
   ```

6. Every time we change the values of `x`, `de`, and `energy`, the program should write those values into the output file specified on the command line[10]. These should be written as three numbers, separated by single spaces, with a `\n` at the end of the line.

7. We can't know in advance how many slices a proton will travel through before its energy is all gone. We just have to look at the energy after each slice, and see if it's still greater than zero[11].

   Near the end of the proton's path, because of the approximations we're making, the `dEdx` function might tell us that the proton loses no energy, even though it has some energy left. That means you also need to check the value of `de` at the end of your "`do-while`" loop:

   ```
   } while ( energy > 0 && de > 0 );
   ```

Once you've written and compiled your program, run it like this to produce an output file to use with your next programs:

```
./simulate 1000 100 100-mev.dat
```

This should produce an output file named `100-mev.dat` containing information about the energy deposited by each proton, in each slice of the patient's body.

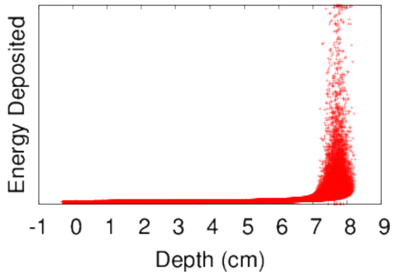


Figure A.28: You can check your first program's results by plotting them with *gnuplot*. This figure shows what you should see if you type:
`plot "100-mev.dat" using 1:2`
It shows the energy deposited in each slice by each proton.

[10] See examples like Program 5.3 in Chapter 5.

[11] This is similar to the `baselpi.cpp` program you wrote for Problem 6 in Chapter 4. In that program, we kept calculating smaller and smaller terms, until we got to one that was less than some limit. That program used a "`do-while`" loop, and we can use one of those here, too.

The file dedx.h, below, contains a function named dEdx for calculating the energy lost (ΔE) in a slice of matter with thickness Δx. This file also contains two random-number functions that we've used before. rand01 generates random numbers uniformly distributed between zero and one, and normal generates random numbers in a Gaussian or "normal" distribution[12].

Program A.8: dedx.h

```
double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }
  return ( rand()/(1.0+RAND_MAX) );
}

double normal () {
  int i, nroll = 12;
  double sum = 0;
  for ( i=0; i<nroll; i++ ) {
    sum += rand01();
  }
  return ( sum - 6.0 );
}

// Returns dE/dx, in MeV * cm^2/g (see units of "constant", below.)
double dEdx (double T, double pmass, double pcharge,
             double rho, double a, double z, double activation) {
  const double constant = 0.1535; // MeV cm^2/g
  const double me = 0.5110034; //MeV/c^2, Electron mass.
  double E, p, beta, gamma, wmax, excite;
  double term1, term2, term3, bbdedx;

  E = T + pmass;
  p = sqrt(T*T + 2.0*pmass*T);
  beta = sqrt(p*p/E/E);
  gamma = 1.0/sqrt(1.0-beta*beta);
  wmax = 2.0*me*beta*beta/(1.0-beta*beta); // MeV
  excite = activation/1.0e6 ; // Convert to MeV.

  term1 = constant*rho*z*pcharge*pcharge/a/(beta*beta);
  term2 = log(2.0*me*gamma*gamma*beta*beta*wmax/excite/excite);
  term3 = 2.0*beta*beta;

  bbdedx = term1*(term2-term3);
  if ( bbdedx < 0.0 ) {
        bbdedx = 0.0;
  }

  // Add 10% gaussian noise:
  bbdedx += 0.1*sqrt(bbdedx)*normal();
  return (bbdedx);
}
```

[12] You can read about both of these in Chapter 11.

Program 2: Visualizing the Results

Your next program will be named **visualize.cpp** and it will help us see how much total energy our beam of protons has deposited at various points along its path. To do this, you'll make what's called a "weighted histogram".

In Chapter 7 we learned about histograms, which are graphs that tell us which values in our data occur most frequently. We imagined dropping marbles into bins to count how many times we'd seen a data value within a particular range.

Think about a similar situation: Imagine you're the owner of a restaurant, and you're concerned about wasting water. You've noticed that sometimes full glasses of water are left on tables after the diners have left. You suspect that some of your wait staff are filling glasses too often. To investigate, you get five large beakers, one for each of your waitpersons. Whenever diners leave, you dump their leftover water into the beaker representing that table's waitperson.

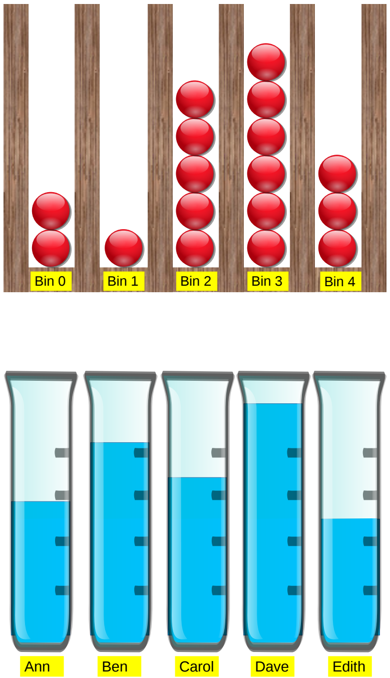


Figure A.29: The top histogram just counts things. The bottom histogram gives each thing a different weight. A weighted histogram doesn't just add 1 for each thing. Instead, it adds some "weight", given by a property that we're interested in (how much water is in a glass, for example). These weights generally won't be integers.

As you can see in Figure A.29, this is almost the same as the histograms we've made before, but instead of putting an integer number of marbles into each bin, we're pouring some (non-integer) amount of water into a beaker. We can think of this as a "weighted" histogram. Instead of just counting each glass as "1 glass", and adding "1" to our histogram, we're giving the glasses different weights, depending on how much water they contain, and adding that weight to the histogram.

In your `visualize.cpp` program, you'll make a weighted histogram that shows how much total energy our proton beam deposited at various points along its path. Your program will be very similar to Program 7.1 in Chapter 7.

Again, to get you started, your physicist friend has taken a break from her busy schedule and written part of the program for you (see Program A.9 below). Notice that she's defined a 100-element array, `hist`, to hold the histogram data. Also notice that this is an array of `double` values instead of integers, since we're making a weighted histogram.

Program A.9: visualize.cpp

```
#include <stdio.h>
#include <stdlib.h>
int main ( int argc, char *argv[] ) {
  const int nbins = 100;
  double hist[nbins];
  FILE *input;
  FILE *output;

  // Gotta go give a lecture. You'll have to
  // write the rest of the program.
}
```

Like the preceding program, this one will expect parameters on its command line, and should complain and exit if it doesn't get the proper number of parameters[13]. Its usage will be:

```
./visualize xmin xmax input output
```

where `xmin` and `xmax` are the minimum and maximum depth we're interested in, in centimeters, `input` is the name of a file produced by your `simulate` program, and `output` is the name of a file into which your program will write the histogram data.

The output file should contain two columns of numbers, separated by a single space. Unlike Program 7.1, the first column here will contain a depth instead of a bin number (see below for instructions about converting bin number to distance). The second column will be the total energy deposited at that depth, in MeV.

To make the histogram, the program should proceed as follows:

1. Make sure the program sets all of the bins to zero at the beginning.
2. Determine the `binwidth`, like this:

   ```
   binwidth = (xmax-xmin)/nbins;
   ```

3. Next, use a `while` loop to read data from the input file. Each line of the file will contain three values: `x`, `de`, and `energy`.
4. Determine which bin this x value belongs in, as Program 7.1 does.
5. Be sure to keep a count of the number of over/underflows, as Program 7.1 does.

Figure A.30: A painting by Gretchen Andrew, from her series "Malignant Epithelial Ovarian Cancer", which aims to "humanize the experience of having cancer".

[13] See Sections 9.15 & 9.16 of Chapter 9.

6. If it's not an over- or underflow, add the value of `de` to this bin. (Note that this is different from Program 7.1, which just adds 1 to the bin.)

7. After processing all of the input data, write the histogram data into the output file. For each bin of the histogram, write two numbers separated by a single space: the depth represented by that bin, and the total amount of energy deposited within it. The depth can be calculated from the bin number, like this:

   ```
   depth = xmin + binwidth*(0.5+i);
   ```
   where `i` is the bin number.

8. Finally, at the bottom of the output file, write a line beginning with a # that tells how many overflows or underflows were seen.

Run your program like this to make a histogram of the data you produced earlier:

```
./visualize  0 10 100-mev.dat hist100.dat
```

You can plot the resulting data file with *gnuplot* like this:

```
plot "hist100.dat" with lines
```

The result should look like Figure A.31.

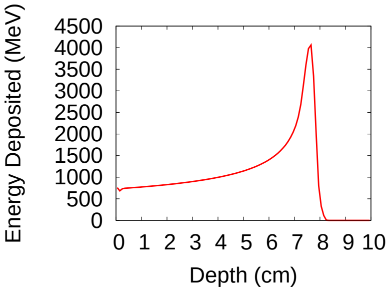


Figure A.31: The total energy deposited at each depth by a 1,000 100-MeV protons.

Program 3: Analyzing the Data

Your last program will be called **`analyze.cpp`**. It will read data produced by your first program and determine how much total energy was deposited in the patient's body, and how much energy was deposited in the tumor.

Like the preceding programs, this one should accept all of its parameters on the command line, and give users a helpful message if they don't give it the right number of arguments. The usage should be:

```
./analyze input tcenter tsize
```

Where "`input`" is the name of a data file produced by your `simulate` program, "`tcenter`" is the depth of the center of the tumor, in cm, and "`tsize`" is the size of the tumor, in cm.

Program A.10: analyze.cpp

```
#include <stdio.h>
#include <stdlib.h>
int main ( int argc, char *argv[] ) {

        // Ack! My lab is on fire (again)!
        // You're on your own here!

}
```

Once again, your physicist friend has written the first part of the program for you, as shown in Program A.10. She didn't have time for much, but you shouldn't have any trouble completing it. Here's how to do it:

Figure A.32: A Russian "Proton" rocket.

Source: Wikimedia Commons

1. First, make sure you define two `double` variables to keep track of the total amount of energy and the amount of energy deposited in the tumor. Make sure both of these are set to zero initially.
2. Next, you'll need to find the depth at which the tumor begins, and the depth at which it ends. These can be found from `tcenter` and `tsize`, like this:

   ```
   xmin = tcenter - tsize/2.0;
   xmax = tcenter + tsize/2.0;
   ```
3. Use a `while` loop to read data from the input file. Each line of the file will contain three values: `x`, `de`, and `energy`.
4. Each time you read a `de` value, add it to the total energy.
5. If `x` is between `xmin` and `xmax`, also add `de` to the amount of energy deposited in the tumor.
6. After reading all of the data, print your results in a nice way that tells the user the total energy and the energy in the tumor. Also tell the user what *fraction* of the total energy was deposited in the tumor, expressed as a percentage. Note that you can tell `printf` to print a percent sign by writing %%.

Figure A.33: The BBC Micro"Proton" computer.

Source: Wikimedia Commons

After you've written your program, run it like this:

```
./analyze 100-mev.dat 8 2
```

Figure A.34: A 2016 "Proton Persona" automobile.

Source: Wikimedia Commons

This tells the program to read the data for 100 Mev protons that you produced with your `simulate` program, and look at the amount of energy that would end up in a two-centimeter-thick tumor located at a depth of eight centimeters. The program's output should look something like this:

```
Total energy deposited: 102252.422287 MeV
Energy deposited in tumor: 28645.976102 MeV
Fraction deposited in tumor: 28.014961%
```

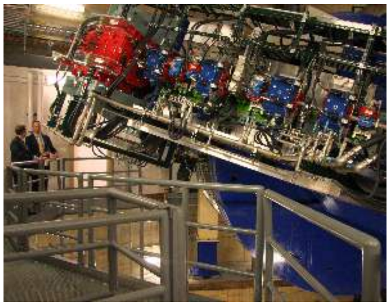


Figure A.35: Proton therapy is a valuable treatment for some types of cancer. It's becoming more widely used, with over 100 treatment centers online now or in planning. Shown above are a facility in Orsay, France (top) and the Mayo Clinic in the US (bottom). The cost, while still significant, is coming down. The ability to minimize radiation damage to surrounding tissues makes it particularly appealing in pediatric cases, where collateral radiation damage can have long-term effects on development.

Results

Using the tools you've written you could find the proton energy that best suits your patient's needs. For example, you could simulate protons of several energies using your `simulate` program:

```
./simulate 1000  50  50-mev.dat
./simulate 1000  75  75-mev.dat
./simulate 1000 100 100-mev.dat
./simulate 1000 125 125-mev.dat
```

then take a look at the energy distribution created by each energy:

```
./visualize 0 10  50-mev.dat hist50.dat
./visualize 0 10  75-mev.dat hist75.dat
./visualize 0 10 100-mev.dat hist100.dat
./visualize 0 10 125-mev.dat hist125.dat
```

You'd see distributions like those shown in Figure A.23 in the introduction. Each distribution has a distinct peak, called the "Bragg peak", near the end of the proton's path. If you saw that one of these peaks lies in the region of the tumor, you might use your `analyze` program to see what fraction of the energy would go into the tumor, like this:

```
./analyze 100-mev.dat 8 2
```

Congratulations, Doctor! You've helped a patient along the road to recovery.

If you're interested in learning more about proton beam therapy, you can find information here:

- Proton Therapy, from Wikipedia.
- The evolution of proton beam therapy: Current and future status, from the NIH's National Center for Biotechnology Information.
- The physics of proton therapy, by Wayne D Newhauser and Rui Zhang.

Project 4: Population Explosion

Introduction

Imagine that a derelict boat washes up on the shore of an uninhabited island. Aboard the boat is a crew of ten rats, all grateful to be on dry land again. Finding plenty of food and water on the island, the happy rats settle down and begin raising families[14].

Boat (1922-1928), Adriano de Sousa Lopes.
Source: Wikimedia Commons

We might wonder how rapidly our rat population grows in their new island home. Common brown Norway rats are known to have a very high reproductive rate of 0.015 offspring per day. In a perfect environment, we might expect their population to grow over time like this:

$$N(t) = N_0 e^{0.015t}$$

where $N(t)$ is the population after t days, given an initial population of N_0. If we graphed the population over a few years, we'd see something like Figure A.36.

This predicts a rat population of 6 trillion trillion after 10 years! Clearly that's unrealistic. Although there are a lot of rats in the world, their total population is probably only a few billion[15].

The problem is that our estimate assumes that birth and death rates will stay the same as the population grows. Observations of the natural world show that this isn't really the case. For example, populations typically share a limited amount of food and other resources. As the population grows, food is harder to find and some individuals die of starvation. Malnutrition also throttles population growth by reducing birth rates. Typically death rates increase and birth rates decline as populations grow. Taking these effects into account, a more realistic graph of our rat population might look like Figure A.37.

This graph shows the population initially increasing, but then levelling

Floating from place to place like this (a phenomenon called "rafting") is one way organisms colonize new territories. About 50 million years ago the first lemurs floated on wind-swept debris across the Mozambique Channel from the African mainland to the island of Madagascar. In 1995, a dozen iguanas floating on trees uprooted by a hurricane colonized the previously iguana-fee Caribbean island of Anguilla.
Sources: Wikimedia Commons and Wikimedia Commons.

[14] This is reminiscent of the famous radio drama Three Skeleton Key, first broadcast in 1949. If you want to hear a scary story, you can listen to it here: mp3 at archive.org

[15] See https://www.worldatlas.com/articles/how-many-rats-are-there-in-the-world.html

off at some constant value. This value (called the *carrying capacity* of the environment) is the population at which the birth rate is equal to the death rate. When these rates are equal, the population no longer increases. The S-shaped curve of this graph is called a *logistic curve* and is typical of the growth of a population colonizing a new, initially resource-rich, environment.

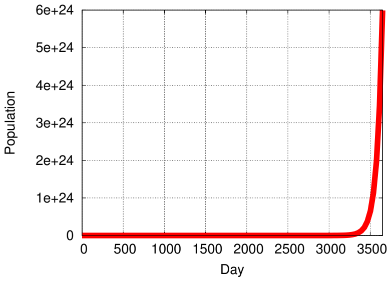


Figure A.36: Rat population given by the equation $N(t) = N_0 e^{0.015t}$.

The Assignment

Now consider a post-apocalyptic scenario where a group of 100 humans is stranded on an island. The island is a pleasant place where the plants and animals could easily provide food and shelter for a population of 1,000 humans. Resigned to their fate, the humans settle down and begin making the best of a bad situation. Ultimately, they have children who grow up knowing no home but the island. These children have grandchildren, and so on down the generations.

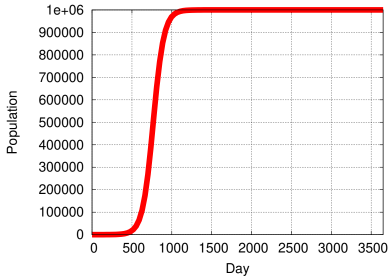


Figure A.37: Rat population with limited resources.

Your task in this project is to write three programs that simulate, visualize, and analyze the growth of such a population.

In order to write a program to model the population's growth, we'll need to know how birth and death rates change as the population increases. The shape of the functions governing birth and death rates will vary from one species to another, and will generally depend on many environmental factors. For the purpose of our simulation, though, let's assume that these rates depend solely on the amount of food available per individual. When food is plentiful, the birth rate is high and the death rate is low. In times of famine, the birth rate is low, and the death rate is high.

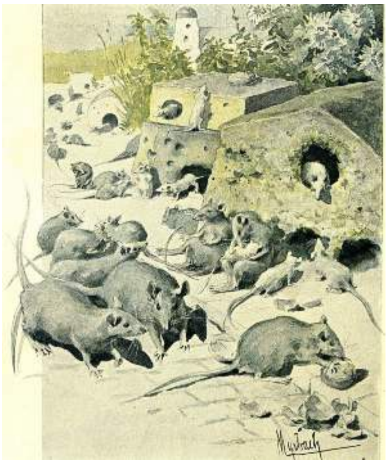


Illustration from Jules Verne's story La Famille Raton, written in 1886.

Source: *Wikimedia Commons*

We'll assume that we're told the total food-producing capacity of the environment, in terms of the number of individuals that can be fully fed. To find each person's share of this bounty (his or her *ration*), we can just divide the total amount of food by the number of people. Birth and death rates will be functions that depend on this ration.

Figure A.38 shows the shapes of the two functions we'll use. These functions give the annual probability of dying or having offspring when the ration has various values. When the ration is 1, everybody is well fed: the annual probability of having offspring is at its maximum, and the probability of dying is at some minimum value due purely to accident, disease, or old age. As the ration approaches zero, the probability of dying approaches 1 (100%) and the probability of giving birth trails off to some tiny value. We'll assume that if the ration is greater than 1, the birth and death rates stay constant at the same values they had when the ration was 1. (We'll ignore any possible ill-effects of overeating!)

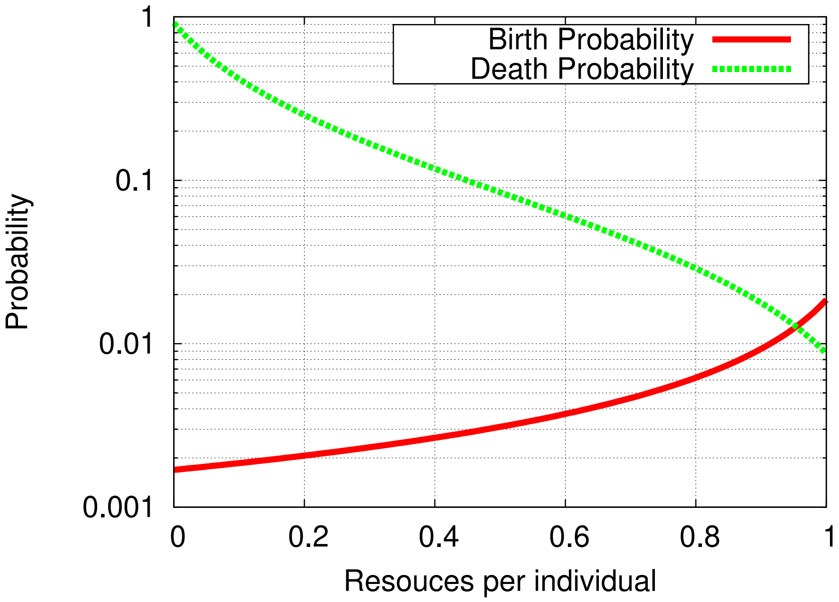


Figure A.38: Annual probability of birth or death as a function of ration.

The birth probability function we've graphed looks like this:

$$b(r) = \begin{cases} \dfrac{b_{max}}{10(1-r)+1} & \text{if } r \leq 1 \\ b_{max} & \text{if } r > 1 \end{cases} \tag{A.1}$$

and the death probability function looks like this:

$$d(r) = \begin{cases} d_{min} + \dfrac{1}{10r+1} - 0.09 & \text{if } r \leq 1 \\ d_{min} & \text{if } r > 1 \end{cases} \tag{A.2}$$

where r is the ration, b_{max} is the maximum probability per year of having offspring, and d_{min} is the minimum probability per year of dying.

Now let's get programming! You'll be writing three programs: **`simulate.cpp`**, **`visualize.cpp`**, and **`analyze.cpp`**. The first will simulate the population's growth, the second will help visualize the results, and the third will do some statistical analysis on them.

Program 1: Simulating Population Growth

Your first job will be to write a program named **simulate.cpp** that simulates the growth of the population over some number of years and writes its results into a file.

Théodore Géricault's *The Raft of the Medusa* (1818-1819).

Source: Wikimedia Commons

Our simulation program's strategy will be this: We'll give the program an initial population, the total amount of food, the values of b_{max} and d_{min}, and tell it how many years to simulate. The program will then loop through the years, one at a time. For each year it will loop through all of the individuals in the population. For each person, the program will check to see whether the person has offspring during that year and whether the person dies during that year, using the $b(r)$ and $d(r)$ functions in Equations A.1 and A.2 above. If the person dies, the population will be reduced by one. If the person has offspring, the population will increase by one[16].

The program should accept all of its parameters on the command line, as described in Section 9.15 of Chapter 9. The usage should be:

```
./simulate population food bmax dmin nyears outfile
```

where:

- `population` is the initial population.
- `food` is the total amount of food the island can produce, in terms of the number of people who can be well-fed.
- `bmax` is b_{max} from Equation A.1 above.
- `dmin` is d_{min} from Equation A.2 above.
- `nyears` is the number of years to simulate.
- `outfile` is the name of a data file into which the program will write its results.

To get you started, I've already written some of the program for you (see Program A.11). All you need to do is complete the program by filling in `main` and the two functions `birthprob` and `deathprob`. Notice that I've added a handy function named `rand01` near the top of the program[17]. It can be used to generate a random number between zero and one.

[16] for simplicity, we're assuming one child per person per year, at most.

[17] This function is described in Section 11.4 in Chapter 11.

Program A.11: simulate.cpp

```
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <stdio.h>

double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }
  return ( rand()/(1.0+RAND_MAX) );
}

double birthprob ( double bmax, double ration ) {
        // Insert function here.
}

double deathprob ( double dmin, double ration ) {
        // Insert function here.
}

int main ( int argc, char *argv[] ) {
  double population;
  double popgrowth;
  int nyears;
  int year;
  int individual;
  double food;
  double ration;
  double bmax, dmin;
  double bprob, dprob;
  FILE *output;

  // Insert program here.
}
```

To complete the program, you'll need to add code to do the following:

1. Check to make sure the user has supplied enough command-line arguments. If there aren't enough command-line arguments, the program should print out a friendly usage message and then stop without trying to do anything else[18].

2. Convert the command-line arguments into the variables `population`, `food`, `bmax`, `dmin`, and `nyears` by using the **`atoi`** and **`atof`** functions. The last command-line argument (the output file name) doesn't need to be converted. You can just use it directly, like this:

[18] See Section 9.16 of Chapter 9 for an example of how to do this.

```
output = fopen( argv[6], "w" );
```

3. Your program will need a pair of nested "`for`" loops: An outer loop that goes through all the years, and an inner loop that goes through all of the individuals in the population and, for each one, checks to see whether that person died or had offspring during the current year.

   The outer loop might start like this:

   ```
   for ( year=0; year<nyears; year++ ) {
   ```

   and the inner loop might start like this:

   ```
   for ( individual=0; individual<population; individual++ ) {
   ```

Crowded Boardwalk, Atlantic City, New Jersey (1910).
Source: Wikimedia Commons

4. At the beginning of each year the program will need to do a few things:

   - Find the `ration` by dividing `food` by `population`
   - Find the probability of having offspring, which we'll call `bprob`, by using the `birthprob` function defined at the top of the program (we'll describe this and the `deathprob` function below).
   - Find the probability of dying, which we'll call `dprob`, by using the `deathprob` function defined at the top of the program.
   - Set `popgrowth` to zero. We'll use this variable to keep track of how much the population grows during the current year. (If there are more deaths than births, this number might be negative, but that's OK.)

5. Inside the inner loop we'll do some things for each individual who's currently in the population:

   - Check to see if that person had offspring during the year. We do this by using the `rand01` function to give us a random number between zero and one, and then checking to see if that number is less than `bprob`. If it is, then we add 1 to `popgrowth`, indicating that a person has been added to the population this year.
   - Similarly, we check to see if the person died this year. We do this by looking to see if `rand01` gives us a number less than `dprob`. If it does, then we subtract 1 from `popgrowth`, indicating that a person has been removed from the population. (Remember that it's OK for `popgrowth` to be negative.)

6. At the end of each year, we add `popgrowth` to `population` to get

the new value for `population`, and we write data about this year into our output file. The values of `year`, `population`, `popgrowth`, `bprob`, `dprob`, and `ration` should be written to the file, in that order, separated by spaces[19].

7. The last step in completing the program is to write the two functions `birthprob` and `deathprob`. The `birthprob` function takes the value of `bmax` and `ration` and uses the relationship shown in Equation A.1 to calculate the birth probability. Similarly, `deathprob` uses `dmin` and `ration` to calculate the death probability, as given by Equation A.2. Note that you'll need an `if/else` statement in each of these functions, to deal with the cases when `ration` is less than one or greater than one.[20]

Edoardo Matania, *Die geschlossene Bank* (1870s).
Source: Wikimedia Commons

After you've completed your program, compile it and run it three times, with these arguments:

```
./simulate 2000 1000 0.0182 0.0077 1000 hipop.dat
./simulate  500 1000 0.0182 0.0077 1000 medpop.dat
./simulate  100 1000 0.0182 0.0077 1000 lopop.dat
```

The three simulations are the same except for the starting population. In the first one, the initial population is higher than the amount of food available in the environment (2,000 people, but only food enough for 1,000). The second simulation has an initial population of 500, with the same amount of food, and the third simulation shows what happens when the initial population is only 100. The values used for `bmax` and `dmin` are actual current worldwide average values for birth and death rates in human populations[21]. Each of the simulations tracks the population growth over a period of 1,000 years.

You can plot your results by giving *gnuplot* the command:

```
plot [0:300] "hipop.dat" with lines, "medpop.dat" with lines, "lopop.dat" with lines
```

which shows just the first 300 years. The result should look something like Figure A.39. Notice that, in all cases, the population eventually settles down to a stable level that's slightly greater than 1,000 individuals.

[19] See Chapter 5 for information about writing data into a file.

[20] See Chapter 3 for information about writing `if/else` statements, and Chapter 9 for information about writing functions.

[21] CIA World Factbook, estimated values for 2018.

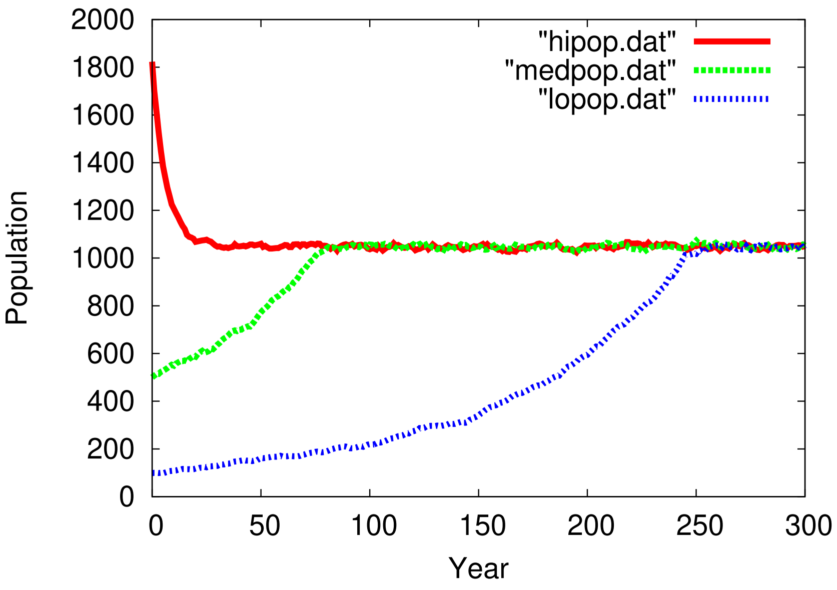


Figure A.39: Population growth when there is sufficient food for 1,000 people, starting with populations of 100, 500, and 2,000 people.

Program 2: Visualizing the Stable Population

So now we know that the island's population always tends toward a particular value, but what is that value exactly? Let's start to investigate this by writing a program to visualize the data from our simulations in a different way. The program will be called **visualize.cpp** and it will let you make graphs like the one shown in Figure A.40. This graph shows population on the horizontal axis, divided into 50 bins. The vertical axis shows how many years had a population within each bin.

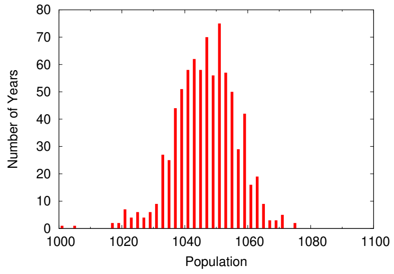


Figure A.40: Histogram of population values from lopop.dat.

This figure is a histogram, like the ones we discussed in Chapter 7. Your program will be similar to Program 7.1 in that chapter. Again, to get you started, I've written part of the program for you (see Program A.12 below). Notice that I've defined a 50-element array, bin, to hold the histogram data.

Program A.12: visualize.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main ( int argc, char *argv[] ) {
  const int nbins=50;
  int bin[nbins];
  double binwidth;
  int binno;
  int overunderflow=0;
  int i;
  FILE *input;
  FILE *output;

  // Insert program here.

}
```

"Cynicus", *The last car for Miramar* (c. 1910).
Source: Wikimedia Commons

Like the preceding program, this one will expect parameters on its command line, and should complain and exit if it doesn't get the proper number of parameters. Its usage will be:

```
./visualize popmin popmax inputfile outputfile
```

where `popmin` and `popmax` are the minimum and maximum population you want to include in your histogram, `inputfile` is the name of a file produced by your `simulate.cpp` program, and `outputfile` is a file into which your new program will write the histogram data.

The input and output files can be opened like this[22]:

```
input  = fopen(argv[3],"r");
output = fopen(argv[4],"w");
```

The output file should contain two columns of numbers, separated by a single space. Unlike Program 7.1, the first column here will contain a population value instead of a bin number (see below for instructions about converting bin number to population). The second column will be the number of years in that bin.

To make the histogram, the program should proceed as follows:

1. First, determine the `binwidth`, like this:

   ```
   binwidth = (popmax-popmin)/nbins;
   ```
2. Next, use a `while` loop to read data from the input file[23]. Each line of the file will contain six values: `year`, `population`, `popgrowth`, `bprob`, `dprob`, and `ration`. The first value is an integer, and the other five are `double`s.

3. Determine which bin this population value belongs in, and increment that bin. Be sure to keep a count of the number of over/underflows, as Program 7.1 does. Since the range of our histogram is `popmin` to `popmax`, the bin number will be:

   ```
   binno = (population-popmin)/binwidth;
   ```

4. After processing all of the input data, write the histogram data into the output file. For each bin of the histogram, write two numbers separated by a single space: the population value represented by that bin, and the value of `bin[i]`. The population value can be calculated from the bin number, like this:

   ```
   population = popmin + binwidth*(0.5+i);
   ```

[22] Notice that we open one file for reading (with `"r"`) and the other for writing (with `"w"`).

[23] See Chapter 5 for information about reading data from files.

where `i` is the bin number.

5. Finally, at the bottom of the output file, write a line beginning with a `#` that tells how many overflows or underflows were seen.

Run your program like this to make a histogram of the data you produced earlier. Start out by looking at population values between 0 and 1,100:

```
./visualize 0 1100 lopop.dat visualize.dat
```

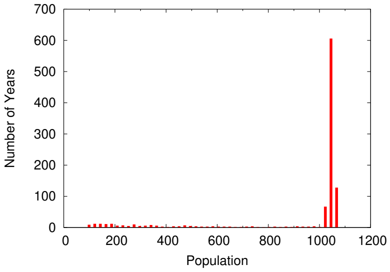


Figure A.41: A histogram of population values from `lopop.dat` in the range 0 to 1,100.

You can plot the resulting data file with *gnuplot* like this:

```
plot "visualize.dat" with impulses lw 5
```

The graph should look like Figure A.41. Now let's zoom in on the region around a population of 1,000 by running your visualize program again, this time setting `popmin` to 1,000 and `popmax` to 1,100:

```
./visualize 1000 1100 lopop.dat visualize.dat
```

You can plot the resulting data file with *gnuplot* like this:

```
plot "visualize.dat" with impulses lw 5
```

The result should look like Figure A.40 at the beginning of this section.

As you can see, the population values cluster around 1,040 or so, slightly above the 1,000 individuals that can be fully fed. Think for a minute about what this means: We're finding that the population tends to settle in at a level where there's not quite enough food to go around. This raises the death rate and lowers the birth rate until the two rates are equal. In your final program you'll find an exact value for this equilibrium population.

Program 3: Finding the Mean and Standard Deviation

Your third program will be named **`analyze.cpp`**. It will read a data file produced by your first program, and give you a statistical summary of the data it contains.

Cualquier parada de cualquier línea de cualquier tranvía de Madrid
PARADA DE TRANVIA
—¡Y sin ver una pareja de guardias!
—¿Cómo quiere usted verla, si hace tres horas que no viene ningún tranvía?

Manuel Tovar Siles, *"Any stop of any line of any tramway of Madrid"* (1920).
Source: Wikimedia Commons

Like the first two programs, `analyze` should accept all of its parameters on the command line, and give users a helpful message if they don't give it the right number of arguments. The usage should be:

```
./analyze popmin popmax inputfile
```

where popmin and popmax delimit the range of population values you're interested in, as they do in your preceding program, and inputfile is the name of a data file produced by your simulate.cpp program.

The output of the analyze program should look like this:

```
Mean population = 1046.245636
Std. dev. = 9.590271
```

Again, I've written some of the program for you (see Program A.13). You'll just need to fill in main.

Program A.13: analyze.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
int main ( int argc, char *argv[] ) {
  int year;
  double population, popgrowth;
  int popmin, popmax;
  double dprob, bprob, ration;
  double sum=0;
  double sum2=0;
  double mean, stddev;
  int nvalues=0;
  FILE *input;

  // Insert program here.

}
```

To analyze the data, the program should proceed as follows:

1. First, open the data file for reading. See Program 7.5 in Chapter 7 for an example of this. Refer to that program to see how to read the data and calculate the average and standard deviation.
2. Like your visualize.cpp program, this new program should use a while loop to read data from the input file. Each line of the file will contain six values: year, population, popgrowth, bprob, dprob, and ration.
3. Unlike Program 7.5, our new program will need to check to see whether a population value is between popmin and popmax before adding it to sum and sum2.

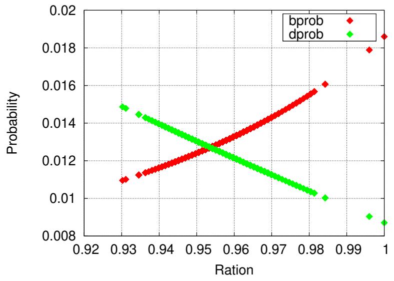


Figure A.42: bprob and dprob versus ration, from our lopop.dat simulation.

If you run your program like this:

```
./analyze 1000 1100 lopop.dat
```

you should see that the mean population value is about[24] 1,046, which corresponds to a ration of $1,000/1,046$ or about 95.6%. If we plot our simulation's birth and death probabilities versus ration, using *gnuplot* commands like this:

```
set xrange [0.92:1]
set yrange [0.008:0.02]
plot "lopop.dat" using 6:4, "" using 6:5
```

(column 6 of our output file is `ration`, column 4 is `bprob` and column 5 is `dprob`) we would see something like Figure A.42. This confirms that birth probability and death probability are equal when the ration is around 95.6%, the ration where our analysis shows that our population is stable.

Thomas Malthus (left) and Charles Darwin.

Conclusion

In 1798, English scholar Thomas Robert Malthus wrote *An Essay on the Principle of Population*, in which he observed that English populations were growing more rapidly than the increase in agricultural production. Malthus anticipated the phenomenon we've explored in this project: Populations tend to grow to the point where resources are no longer sufficient for everyone, causing death rates to increase and birth rates to decline until the population stabilizes. Malthus's ideas about competition for scarce resources were an inspiration for Charles Darwin's theory of evolution by natural selection.

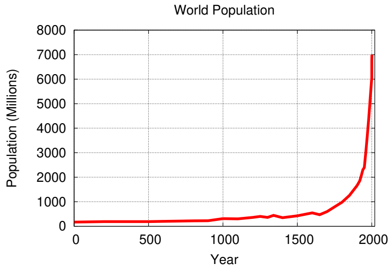


Figure A.43: World population growth.
Source: *Wikimedia Commons*

Such plateaus in population have occurred many times in human history, but have typically been temporary and limited to a geographic region. In Malthus's time, for example, England was heading for a shortage of food, while Russia had an overabundance of agricultural capacity. Malthus expected these shortages to last only until new agricultural land had been developed, or until improvements in agriculture increased the yield of existing land.

Arnold Böcklin, *The Isle of the Dead*, third version (1883) and *The Isle of the Living* (1888).
Source: *Wikimedia Commons* and *Wikimedia Commons*

Globally, the human race has shown no slowing of its exponential growth rate (see Figure A.43). So far, development of new land and improvements in agricultural science have, on average, kept ahead of population growth, but humans also depend on fresh water, shelter, and other limited resources. Some people estimate[25] that the global

[24] The value you see will vary, because the simulation uses random numbers

[25] See *https://en.wikipedia.org/wiki/Planetary_boundaries*.

population already exceeds the Earth's carrying capacity[26].

What will happen to our islanders? Will they find a clever way to avoid a "Malthusian crisis?" Let's wish them luck, and the same for the inhabitants of this island Earth.

[26] Apparently we're not running short of physical space. John Brunner's novel Stand on Zanzibar notes that 7 billion people (the current population of Earth) could fit on the island of Zanzibar – if they stood shoulder to shoulder!

Project 5: Yard Sale!

Introduction

Every August a 630-mile-long yard sale stretches from Michigan to Alabama along US Highway 127. It's called the "World's Longest Yard Sale". Thousands of people visit it. In the early 21^{st} century Economists began to realize that yard sales like this provide a good model for the whole world's economy. By simulating the interactions between buyers and sellers at such a sale, we can make predictions about wealth distribution that match data observed in the real world. The trick is to assume that the economy is made up of many, many one-to-one interactions where a buyer and a seller exchange some wealth.

Source: Wikimedia Commons

Source: Wikimedia Commons

Economists gauge a person's wealth by looking not just at how much money you have, but also the value of the goods you own. Imagine that I'm a vendor at the yard sale and you're a shopper. If you pay me five dollars for a toaster, an economist would traditionally have said that there was no net change in either person's wealth: I have your five dollars, but you now have a toaster worth five dollars.

But is it really? What if, when you get home, you find that the toaster doesn't work. Then you really have a toaster worth less than five dollars, but I still have your money. We could say that you've lost some wealth by giving me five dollars and getting something worth less than that, and I've gained some wealth by getting five dollars in exchange for a worthless toaster. In that case, wealth has flowed from you to me, making you poorer and me richer.

This happened because you mis-judged the value of the toaster. Traditionally, economists have assumed that shoppers are good at judging the value of things, and economic models have used this assumption to make predictions about the economy. But recently economists have become interested in models that take into account the fact that buyers and sellers often make mistakes about the value of things. A seller might sell a "worthless" painting for five dollars, only to find later that it's a valuable Picasso, or a buyer might pay a lot of money for a "Rolex" watch only to find that it's a cheap knock-off.

Anirban Chakraborti, who first proposed the "yard sale" model of economics in 2002.

The mistakes we make are usually small, but we probably always make some small error when we assign a value to something we buy or sell. The effect of this is that wealth flows around the economy, with some people becoming more wealthy than others. If everyone had an equal chance of gaining or losing an equal amount because of these mistakes, we might assume that, on average, they don't matter, and that any inequalities of wealth would even out over time. But the yard sale models that Economists have developed, and which match real-world economic data, make an additional assumption: They assume that the biggest possible mistake in each transaction is the total wealth of the *poorest* person involved in the transaction. (A person with only one dollar can't buy the five-dollar toaster, no matter whether the toaster is broken or not.) By doing this we're ignoring people who win the lottery or accidentally sell a Picasso for five dollars, but it turns out that those situations are rare and don't have much effect on the economy as a whole[27].

In this project we're going to write three programs that investigate such a yard-sale model of the economy. The first program (**`simulate.cpp`**) will simulate lots of interactions between buyers and sellers. The second (**`visualize.cpp`**) will visualize the distribution of wealth after some time has passed. The third (**`analyze.cpp`**) will analyze the data and boil it down to a single number that measures how evenly wealth is distributed. Let's get started!

Program 1: Buyers and Sellers

Our first program will be named **`simulate.cpp`**, and it should start out like Program A.14 below. The program will simulate many random transactions between pairs of people, and track the wealth flowing from person to person. We'll assume everybody starts out with the same amount of wealth.

How the Program Works

The program should accept three parameters on the command line: The initial wealth of each person, the number of transactions we want

[27] These models also assume that wealth of any kind can be exchanged. I can pay you five dollars for that toaster, or I can trade you a record player for it. My wealth includes both the money I have and the value of the items I own.

Program A.14: simulate.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main ( int argc, char* argv[] ) {
  const int N = 10000;  // Number of people.
  double wealth[N];     // Wealth of each person.
  double wstart;        // Starting wealth of each person.
  double mistake;       // Size of a mistake.
  double flip;          // A random number, used for deciding who made the mistake.
  double ntransactions; // Number of transactions.
  int alice, bob;       // The two people involved in a transaction.
  int poor;             // which of the two people is poorer.
  int i;
  FILE *output;

  srand( time(NULL) );  // Set the seed of the random number generator.

  // Put the rest of the program here!

}
```

to simulate, and the name of a file we want to write our results into. For example:

```
./simulate 100 2e+5 output.dat
```

The first number is the initial wealth of each person, the second is how many transactions we want to simulate[28], and the final argument is the name of the output file we want to create.

The program should assume that this is a very big yard sale, with 10,000 people swapping money and goods. That's the population of a small town or a rural county. To keep track of how much wealth each person has, it should use an array with 10,000 elements. The wealth of person number `i` will be `wealth[i]`. A person's wealth will generally be a number with decimal places, so `wealth` will need to be an array of `double`s.

We'll start each person out with the same amount of wealth. Let's call it `wstart`. After setting the initial wealths, the program should enter a loop that simulates some number of random transactions. For each transaction, we'll pick two people at random. Let's call them `alice` and `bob`, and their wealths will be `wealth[alice]` and `wealth[bob]`.

After we've picked two people, we need to see which one is poorer by comparing their wealths. Let's have another variable, `poor`, and say

[28] This number is in C-style scientific notation. In this example, we've used `2e+5` which is 2×10^5, or 200,000. See Section 4.3 of Chapter 4.

that if Alice is poorer, `poor=alice` and if Bob is poorer, `poor=bob`.

Now assume that somebody makes a mistake in the transaction. Remember that we're limiting the size of the mistake to the wealth of the poorer person, so at most the mistake will be `wealth[poor]`. Let's say that the size of the mistake is a random number between zero and one, multiplied by `wealth[poor]`.

Then we "flip a coin" to decide which person, Alice or Bob, benefits from this mistake. We do this by generating a random number between zero and one. If this number is greater than 0.5 Alice wins, otherwise Bob wins. If Alice wins, the amount of the mistake is added to her wealth and subtracted from Bob's wealth. If Bob wins, the mistake is added to his wealth and subtracted from Alice's.

Aaah, wealth! (Portuguese actor António Silva portraying a wealthy man)
Source: Wikimedia Commons

After the program has done the requested number of transactions, it should write the final wealth of each person into the file specified on the command line. The output file should have two columns separated by a space: person number and the wealth of that person.

How to Write the Program

To get you started, Program A.14 shows part of `simulate.cpp`. It includes all of the variables you'll need. You just need to write the middle part, where all the work gets done. To complete the program, you'll need to add code to do the following:

1. Check to make sure the user has supplied enough command-line arguments. If there aren't enough command-line arguments, the program should print out a friendly usage message and then stop without trying to do anything else[29].

2. Convert the command-line arguments into the variables `wstart` and `ntransactions` by using the **`atof`** function[30].The last command-line argument (the output file name) doesn't need to be converted. You can just use it directly, like this:

   ```
   output = fopen( argv[3], "w" );
   ```

3. Next you'll need a "`for`" loop to set the initial wealth of each person to `wstart`.

4. Then you'll need a second "`for`" loop that goes through `ntransactions`

[29] See Section 9.16 of Chapter 9 for an example of how to do this.

[30] Notice that we've chosen to make `ntransactions` a `double`, even though it will always have some integer value. That's because we'll be using large numbers of transactions, and it's convenient to write things like `1e+7` instead of `10000000`, so we don't have to carefully count zeros. C only lets you use scientific notation with `double`s.

transactions. During each transaction the program will need to do several things:

(a) Pick two random people to be Alice and Bob for this transaction. You might do something like this:

```
    alice = rand()/(1.0+RAND_MAX) * N;
    bob   = rand()/(1.0+RAND_MAX) * N;
```

Notice that this generates a random number between zero and (almost) one, and then multiplies it by N, the total number of people [31].

(b) Then we need to use an "if" statement to check which person has the smaller wealth. Set the variable poor to equal either alice or bob, as appropriate.

(c) Next the program needs to determine a random size for the mistake that's made in this transaction. Remember that it should be an amount between zero and wealth[poor]. One way to do this is:

```
    mistake = wealth[poor]*rand()/(1.0+RAND_MAX);
```

(d) As the last thing in the loop the program should "flip a coin" to see whether Alice or Bob gets the benefit of the mistake. To do this, generate a random number between zero and one, and then use an "if" statement to see if it's greater than 0.5. If it is, then Alice wins. Transfer mistake amount of wealth from the loser to the winner.

5. After the loop is done, the program should write its results into a file[32]. This should be done with a third "for" loop. For each person, there should be one line in the file with two numbers separated by a space. For person "i" the numbers should be i and wealth[i].

Running the Program

After you've created the program, run it several times to make some output files that you'll use with the next two programs. Try these commands:

```
./simulate 100 0 simulate-0.dat
./simulate 100 1e+4 simulate-10K.dat
./simulate 100 1e+6 simulate-1M.dat
./simulate 100 1e+9 simulate-1G.dat
```

[31] On rare occasions, at random, it will turn out that "Alice" and "Bob" are the same person, but we won't worry about that. It happens rarely, and it won't affect the results.

[32] See examples like Program 5.3 in Chapter 5.

Those commands will create four output files representing a starting wealth of $100 for each person, and simulating 0 transactions, 10 thousand transactions, then 1 million and 1 billion transactions. If you look inside any of these files with *nano* you should see two columns of numbers. The first column will be the person number (an integer) and the second column will be that person's wealth (a number with decimal places) after the specified number of transactions. You can graph the results with *gnuplot* if you like, using *gnuplot* commands like:

```
plot "simulate-1M.dat" with impulses
```

You should see graphs like the ones in Figure A.44.

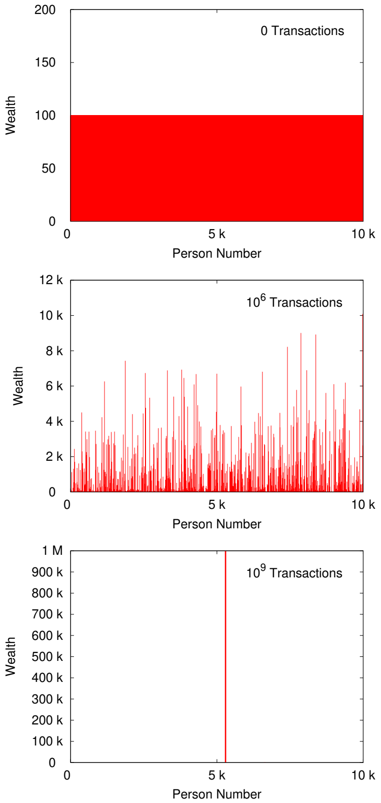


Figure A.44: The top graph shows the distribution of wealth after 0 transactions. Everybody has the same amount of money ($100). The middle graph shows the situation after 1 million transactions. Now some people have a lot more wealth than others. The bottom graph show the situation after 1 billion transactions. Now one random, lucky person has *all* of the money, and everyone else has nothing!

Look at what happens as the number of transactions increases. At zero transactions everybody has the same amount of money. After a million transactions wealth has spread around, and some people have thousands of dollars. This isn't too surprising. But after a billion transactions we find that one lucky person has *all* of the money, and nobody else has any! If you run this billion-transaction simulation several times, you'll find that one person always ends up with all the money, but it will be a different person each time.

That's something that economists have found to be an inescapable property of the yard sale model: If you let it run long enough one person inevitably ends up with all the wealth.

Program 2: Visualizing at the results

Let's take a closer look at how our simulation distributes wealth. To investigate this, we might make a graph that shows wealth across the bottom, divided into equal-sized ranges like $0-$25, $25-$50, $50-$75, and so on. On the vertical axis we could show how many people have a wealth in each range. We learned in Chapter 7 that a graph like this is called a histogram.

The next program you'll write is named **`visualize.cpp`** and it will make histograms of the simulated wealth data created by your first program. The new program will be similar to Program 7.1 in Chapter 7. It should start out like Program A.15 below.

Program A.15: visualize.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main ( int argc, char *argv[] ) {
  const int nbins=100;
  int bin[nbins];      // How many people are in each wealth range.
  double binsize;      // Width of wealth ranges.
  int n;               // "Person number".
  double wealth;       // Wealth of that person.
  double maxwealth;    // Maximum wealth we want to graph.
  int binno;           // Bin number for a person, based on person's wealth.
  int overunderflow=0; // How many people were outside the range of the graph?
  int i;
  FILE *input;
  FILE *output;

  // Insert program here.

}
```

How the Program Works

Like the preceding program, this one will expect parameters on its command line, and should complain and exit if it doesn't get the proper number of parameters. Its usage will be:

```
./visualize maxwealth input.dat output.dat
```

where `maxwealth` is the maximum wealth you want to include in your histogram, `input.dat` is the name of a file produced by your `simulate.cpp` program, and `output.dat` is a file into which your new program will write the histogram data.

The output file should contain two columns of numbers, separated by a single space. Unlike Program 7.1, the first column here will contain a wealth value instead of a bin number (see below for instructions about converting bin number to wealth). The second column will be the number of people who have that amount of wealth.

Postcard: "Youth poverty at the beginning of the 20th century in Europe."
Source: Wikimedia Commons

How to Write the Program

To make the histogram, the program should proceed as follows:

1. Check to make sure the user has supplied enough command-line arguments. If there aren't enough command-line arguments, the

program should print out a friendly usage message and then stop without trying to do anything else.

2. Convert the first command-line argument into the variable `maxwealth` by using the **`atof`** function. The other two command-line arguments (the input and output file names) don't need to be converted. The input and output files can be opened like this[33]:

   ```
   input  = fopen(argv[2],"r");
   output = fopen(argv[3],"w");
   ```

3. Next, determine the `binwidth`, like this:

   ```
   binwidth = maxwealth/nbins;
   ```

4. Use a "`for`" loop to set all the elements of `bin` to zero.

5. Now use a `while` loop to read data from the input file[34]. Each line of the file will contain two values: A person number and that person's wealth. The first value is an integer, and second is a `double`.

6. Determine which bin each person's wealth value belongs in, and increment that bin. Be sure to keep a count of the number of over/underflows, as Program 7.1 does. Since the size of each bin is `binwidth`, the bin number will be:

   ```
   binno = wealth/binwidth;
   ```

7. After processing all of the input data, write the histogram data into the output file. For each bin of the histogram, write two numbers separated by a single space: the first number is the wealth value represented by that bin, and the second is the value of `bin[i]`. The wealth value can be calculated from the bin number, like this:

   ```
   wealth = binwidth*(0.5+i);
   ```

   where `i` is the bin number. This will give you the wealth at the midpoint of that bin's wealth range.

8. Finally, at the bottom of the output file, write a line beginning with a # that tells how many overflows or underflows were seen.

After you've written the program, run it a few times like this to create histograms from the files you created previously, limiting the graph to

[33] Notice that we open one file for reading (with `"r"`) and the other for writing (with `"w"`).

[34] See Chapter 5 for information about reading data from files. In particular, look at Program 5.4.

a maximum wealth of $2,500:

```
./visualize 2500 simulate-0.dat visualize-0.dat
./visualize 2500 simulate-10K.dat visualize-10K.dat
./visualize 2500 simulate-1M.dat visualize-1M.dat
./visualize 2500 simulate-1G.dat visualize-1G.dat
```

You can use *gnuplot* to view the histograms by giving it commands like:

```
set log y
set yrange [0.1:]
plot "visualize-10K.dat" with impulses
```

This will draw a vertical line for each wealth range, with the height of the line indicating the number of people who have a wealth in that range. The first command makes the Y-axis logarithmic. If we didn't do this, we wouldn't be able to the bins that only have a few people in them. Your graphs should look something like the ones shown in Figure A.46.

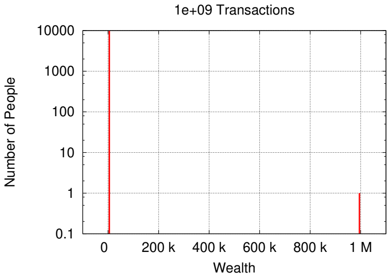


Figure A.45: Wealth distribution after 1 billion transactions, showing wealth up to $1 million.

You can see that the data in the last graphs is starting to run off the right-hand edge of the graph. The total amount of money in our population is $1 million ($100 per person × 10,000 people). Let's graph the data from our longest simulation using this as `maxwealth`. To do that, run your `visualize` program again, like this:

```
./visualize 1000000 simulate-1G.dat visualize-long-1G.dat
```

This extends the wealth scale out to $1,000,000. If you graph the new file with *gnuplot* (again using a logarithmic Y-axis) you should see something like Figure A.45.

This is another way of seeing that only one person ends up with all of the money. The short spike on the right-hand side represents the one person who now has 1 million dollars. The tall spike on the left-hand side is everyone else, with zero dollars[35].

Program 3: Quantifying Wealth Inequality

Our simulated economy produces severe wealth inequality, but how does it compare to real-life economies? How can we measure the amount of wealth inequality? One way of quantifying it is called the "Gini Coefficient[36]".

[35] Sometimes after a billion transactions you'll find that there are still two people who have some money. After more transactions, though, one of them always ends up with all of the money.

[36] See https://en.wikipedia.org/wiki/Gini_coefficient.

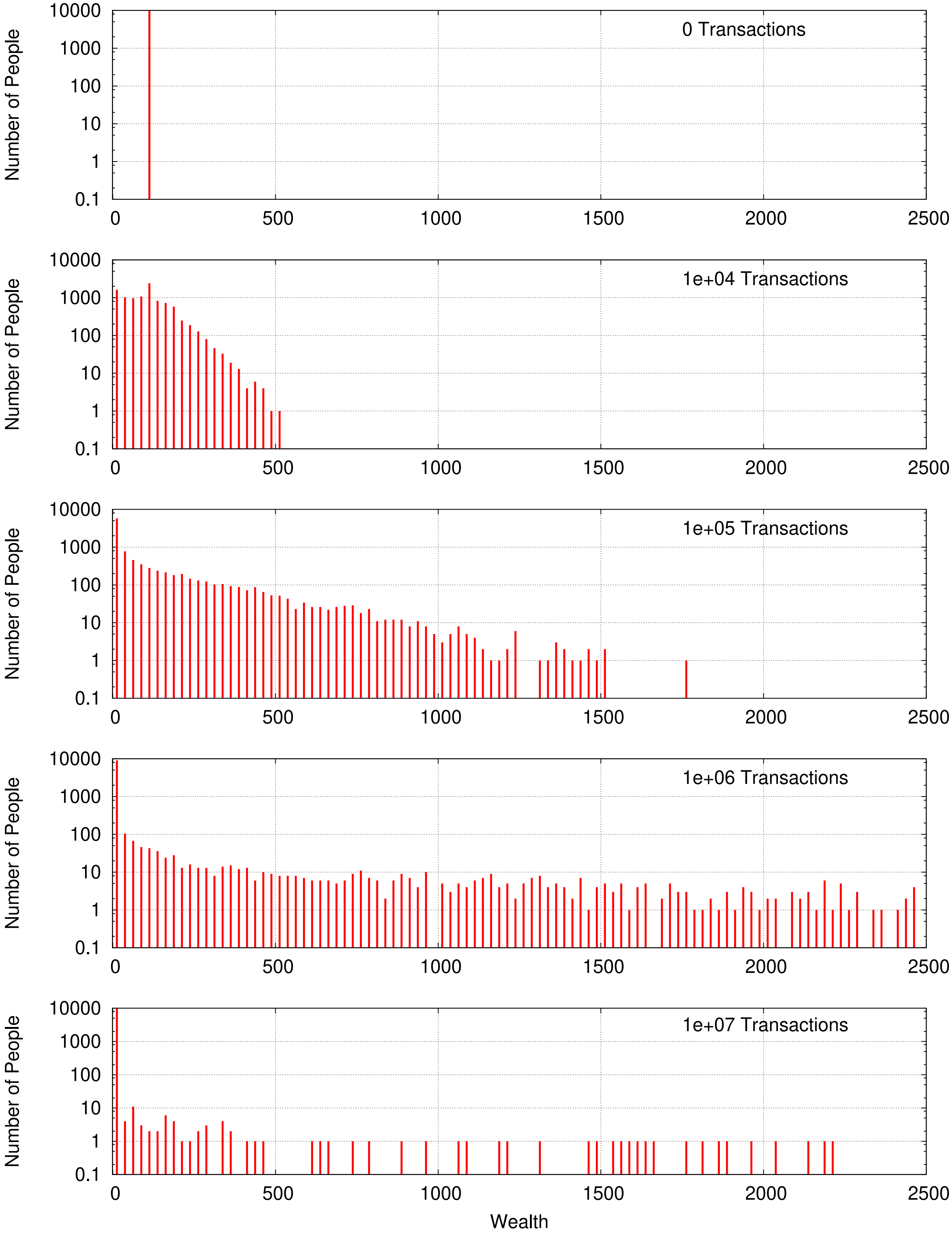


Figure A.46: Histograms of wealth after different numbers of transactions.

The Gini Coefficient starts by measuring the average difference in wealth between any two individuals in the population. (It ignores the sign of this difference by taking the absolute value.) Then it divides the result by the total amount of wealth in the population. A Gini Coefficient of zero corresponds to an economy where everybody has the same amount of wealth. A value of one corresponds to an economy where a single person has all the wealth, and everyone else has nothing. Real-life economies fall somewhere between these two extremes.

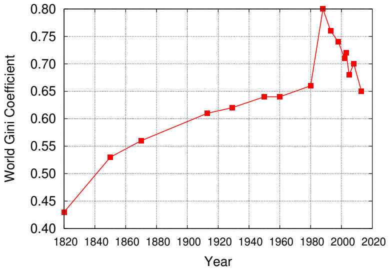


Figure A.47: Estimated world-wide Gini Coefficient, by year. See Milanovic and World Bank in the "Further Reading" section below.

Researchers at the World Bank have estimated values for the world-wide Gini Coefficient for various years, beginning with 1820 (see Figure A.47). The value seems to have risen to a peak of about 0.8 in the 1980s and then begun a downward trend. The current value is about 0.65[37]. Your third program, **`analyze.cpp`**, will read the data produced by your simulation and calculate the Gini Coefficient for your simulated economy.

How the Program Works

Like the first two programs, this one should accept arguments on the command line. In this case, there will be just one argument: the name of a data file produced by your `simulate.cpp` program. For example, you should be able to run your latest program like this:

```
./analyze simulate-10K.dat
```

Your program should start by reading the data from the data file and putting it back into a 10,000-element array called `wealth`, just like the array you used in your first program.

Next your program will need to add up the total wealth of all of the people. You'll need this later for calculating the Gini Coefficient.

"Children sleeping in Mulberry Street" (detail), by Jacob Riis (1890).
Source: Wikimedia Commons

After that, you'll need to go through each pair of people in the population, find the difference in their income, and add its absolute value to a sum. You should do this with two nested "`for`" loops. Once the wealth differences have all been added up, you can use that sum and the total wealth to calculate the Gini Coefficient. Mathematically, the Gini Coefficient is defined as:

$$\texttt{gini} = \frac{\sum_{\texttt{i}} \sum_{\texttt{j}} |\,\texttt{wealth[i]} - \texttt{wealth[j]}\,|}{2\texttt{N} \sum_{\texttt{i}} \texttt{wealth[i]}}$$

[37] Note that some writers refer to the "Gini Index", which is just 100 times the Gini Coefficient. That means the current world-wide Gini Index is about 65.

Program A.16 below shows how your program should start. It contains all the variables you'll need. You just need to fill in the rest of the program.

Program A.16: analyze.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main ( int argc, char* argv[] ) {
  const int N = 10000;  // Number of people.
  double wealth[N];     // Wealth of each person.
  double sum = 0;       // Sum of all the wealth.
  double sumdiff = 0;   // Sum of wealth differences between pairs of people.
  double gini;          // Gini coeffficient.
  int n;                // ``Person number''
  int i,j;
  FILE *input;

  // Add the rest of the program here.
}
```

How to Write the Program

1. Check to make sure the user has supplied enough command-line arguments. If there aren't enough command-line arguments, the program should print out a friendly usage message and then stop without trying to do anything else.
2. The only command-line argument (the input file name) doesn't need to be converted. You can just use it directly, like this:

   ```
   input = fopen( argv[1], "r" );
   ```
3. Next you'll need a "for" loop that repeats 10,000 times (the value of N in the program) and reads one line out of the input file each time. The input file has two columns of data: the person number and that person's wealth. That means you should have a statement like this for reading a line from the input file:

   ```
   fscanf( input, "%d %lf", &n, &wealth[i] );
   ```
   As you read each wealth value, add it to the value of sum. This will give you the sum of all the wealth in the population, which you'll need later for calculating the Gini Coefficient.
4. Now the program needs to find the difference in wealth between each pair of people in the population. To do this you'll need a pair of nested "for" loops. Use the fabs function to get the absolute value of the wealth difference, and then add it to sumdiff like this:

   ```
   sumdiff += fabs( wealth[i] - wealth[j] );
   ```

Note that this will actually count each pair of people twice. For example, if `i` is 20 and `j` is 30, the sum will include both `wealth[20] - wealth[30]` and `wealth[30] - wealth[20]`. We'll take care of this later by dividing `sumdiff` by 2 when doing the Gini Coefficient calculation.

5. Finally, the program just needs to calculate the Gini Coefficient and print it out. The Gini Coefficient will be equal to `sumdiff/( 2.0*N*sum )`.

The Gini Coefficient calculated by your program will be a value between zero and one. If you run it with your simulation data for 10,000 transactions, like this:

```
./analyze simulate-10K.dat
```

you should see a Gini Coefficient of about 0.37. If you run it with the simulation data for 1 billion transactions, the value should be much higher, almost 1.0. Figure A.48 shows how the Gini Coefficient varies with the number of transactions. As you can see, it approaches a value of one for large numbers of transactions, meaning that only a few people end up with all of the wealth.

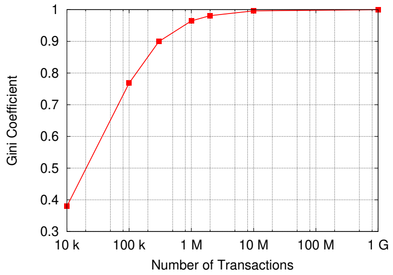


Figure A.48: Gini Coefficient calculated for various numbers of transactions using our yard sale simulation.

Conclusion

So what does this model of economics tell us about the real world? Although there is great inequality of wealth (for example, five billionaires now have more wealth than the poorest half of humanity combined), it seems unrealistic that one person would end up with all of it.

The yard sale model seems pretty simple. It just makes two assumptions: pairs of people exchange wealth, and poor people can't spend more money than they have. Why does it make a prediction that's so different from what we see in the world around us? Clearly there's some factor that we're leaving out of our model.

It might seem that everybody at the yard sale has an equal opportunity to gain wealth, and at first they do. Initially wealth is distributed evenly among all of them, with perfect symmetry. But this initial symmetry is spontaneously broken as soon as some people become a little poorer than others. Poorer people in the model are always at an economic disadvantage because poverty limits the size of the economic risks they can take. This creates a tendency for the rich to get richer and the poor to get poorer, causing the yard sale model to inevitably collapse into oligarchy.

Why doesn't this happen in the real world? Mathematician Bruce Boghosian at Tufts University and his economist colleagues have shown that by transferring a small fraction of wealth from rich people to poor people after each transaction, the yard sale model's wealth distribution can be stabilized. In the real world, this corresponds to the wealth redistribution that's done by taxes and social programs.

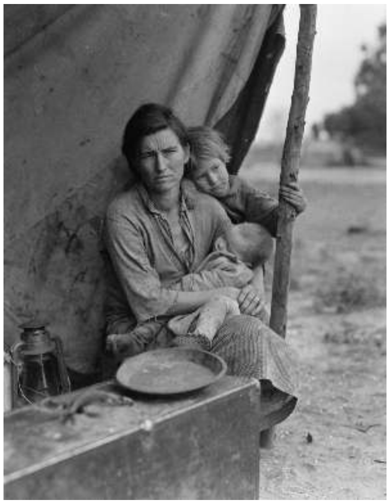
Lou Hoover, First Lady of the United States, with her sons.

With this one small change, Boghosian has found the modified yard sale model can match recent European and U.S. wealth distribution patterns to within 2%. By making two more tweaks, allowing people to go into debt and accounting for advantages that wealthier people have in business transactions, the model can match U.S. data over a span of several decades with an accuracy of a fraction of a percent.

Boghosian also points to economies where social programs have broken down, like Armenia after the fall of the Soviet Union. In those cases, the economy really does devolve into oligarchy, with all of the wealth being held by a few people after an initial struggle for resources, just as our unmodified yard sale model would predict.

In a 2019 *Scientific American* article Boghosian said

> "We find it noteworthy that the best-fitting model for empirical wealth distribution discovered so far is one that would be completely unstable without redistribution rather than one based on a supposed equilibrium of market forces. In fact, these mathematical models demonstrate that far from wealth trickling down to the poor, the natural inclination of wealth is to flow upward, so that the 'natural' wealth distribution in a free-market economy is one of complete oligarchy. It is only redistribution that sets limits on inequality."



Migrant Mother, Photo by Dorothea Lange.
Source: Wikimedia Commons

Further Reading

- *"The Mathematics of Inequality"*, https://now.tufts.edu/articles/mathematics-inequality.
- Bruce M. Boghosian, *"Is Inequality Inevitable?"* (originally published under the title *"The Inescapable Casino"*), Scientific American 321, 5, 70-77 (November 2019).
- Anirban Chakraborti, *"Distributions of money in model markets of economy"*, https://arxiv.org/abs/cond-mat/0205221.
- Branko Milanovic, *"Global Inequality and the Global Inequality Extraction Ratio"*, http://documents1.worldbank.org/curated/en/389721468330911675/pdf/WPS5044.pdf.
- World Bank, *"Poverty and Prosperity 2016 / Taking on Inequality"*, https://openknowledge.worldbank.org/bitstream/handle/10986/25078/9781464809583.pdf.

Project 6: Virus

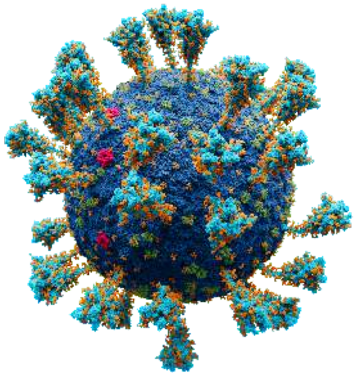
Introduction

Imagine a time in the future when a new virus, deadlier than COVID-19, is ravaging the country. You're a health-care professional working on the island of Natucket, off the coast of Massachusetts. So far, no one on Natucket has contracted the virus, but you're a good computer programmer and your boss has asked you to simulate the potential effect of a virus outbreak on the island's residents.

Nantucket, Massachusetts.
Source: Wikimedia Commons

Nantucket has a full-time population of a little over 10,000. During the tourist season it normally swells to 50,000, but concerns over the virus and restrictions on travel have kept visitors away this year. Nonetheless, if the disease reaches Nantucket the number of patients could easily exceed the twenty beds available in the island's only hospital.

Fortunately, a very effective vaccine has been developed. One thing your boss has asked you to look into is how different percentages of vaccinated people would affect the impact of the virus.



The SARS-CoV-2 virus, which causes COVID-19.
Source: Wikimedia Commons

You also know that measures like mask-wearing and social distancing can slow the spread of the virus by reducing the probabilty that it will be transmitted from one person to another. This is the second thing your boss has asked you to investigate: How does transmission probability influence the spread of the disease?

It's late at night now, and you've started writing three programs that will help answer these questions. Go get some sleep and finish them tomorrow!

Program 1: Simulating The Progress of the Disease

Your first program is called **`simulate.cpp`** (see Program A.17 on page 572). It tracks the health of 10,000 people (the same as Nantucket's population) for 100 days. Each person's health status is represented by one of the numbers shown in Table A.1. The program uses a 10,000-element array of integers to keep track of the status of each person. The program assumes that 100 people (1%) are infected on the first day.

Status	Number	Symbol
Susceptible	0	`is_susceptible`
Recovered	-1	`is_recovered`
Vaccinated	-2	`is_vaccinated`
Dead	-3	`is_dead`
Infected	Any positive number: # of days infected	

Table A.1: Numerical status indicators for each person in the simulation. The last column shows symbols (defined in `virus.h` – see below) that we can use instead of numbers if we like. Notice that any positive number indicates the number of days that an infected person has been sick.

Each day, the program simulates random interactions between people. These interactions cause some susceptible people to get sick. Each day, sick people have some chance of getting well or dying. At the end of each day, the program counts how many people are in each of these states and writes that data into a file.

The program should accept three arguments on the command line: The fraction of people who are vaccinated (`vprob`), the probability of catching the disease from an infected person (`tprob`), and the name of the output file.

By running the program with various values of `vprob` and `tprob` you can investigate the effects of vaccination and masking or social distancing on the spread of the disease, as you boss asked you to do.

Vaccination and mask-wearing can slow the spread of some diseases.
Sources: Wikimedia Commons and Wikimedia Commons

How the Program Works

Your program actually has *two* 10,000-element arrays: one to hold each person's current health status, and the other to hold the status they'll have during the next day of the simulation. At the end of each day, the data from the "next day" array should be copied into the "current status" array.

The program should loop through all 100 days, and on each day it should loop through all 10,000 people, determining each person's new status. The new status will be determined by two functions you need to write.

To decide whether a susceptible person has been infected, you should write a function named **`catch_or_not`** that simulates a random number of encounters between the person and other people in the population. When an infected person is encountered, the function flips a virtual coin to see if the susceptible person catches the infection.

To decide whether an infected person gets well, dies, or stays sick, you should write another function named **`die_or_not`**. If the person survives more than about 14 days, this function determines that the person has recovered and is no longer sick or susceptible to the infection.

The program should assume that vaccinated people are perfectly protected, and have no chance of becoming infected[38].

How to Write the Program

Your previous day's work is shown in Program A.17. You'll just need to finish the two functions at the top, and finish the middle of the main program. Follow the instructions below to complete the program. Note that the program also uses two header files that you wrote the previous day (Wow, you were really productive!), named `random.h` and `virus.h`. You'll find these in the "Header Files" section at the end of this project.

The program contains two arrays named `status` and `newstatus`. These are used to keep track of each person's current status and the new status they'll have on the next day of the simulation.

Start by writing the `catch_or_not` and `die_or_not` functions.

Writing `catch_or_not`:
This function will simulate a random number of interactions between a susceptible person and other people in the population and return an integer that's the new status for the person (one of the numbers in Table A.1). This function takes three arguments: `tprob`, the transmission probability; `npeople`, the number of people in the population; and `status`, the array of current status information. To write this function, follow these steps (and define any variables you need for them):

The island of Nantucket, once famous for whaling. Herman Melville said "Two thirds of this terraqueous globe are the Nantucketer's. For the sea is his; he owns it, as Emperors own empires".
Source: Wikimedia Commons

1. You're first going to need to generate a random number of people that this person is exposed to during the day. The header file `random.h` contains a function that will be handy for this, named `rand01`. It generates random numbers that are uniformly dis-

[38] This is obviously not the case for a real vaccine, but it makes our program simpler. The Pfizer vaccine is 95% effective at preventing hospitalization due to Covid-19, so 100% effectiveness for a vaccine is not an unreasonable approximation for purposes of our simulation.

Program A.17: simulate.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "random.h"
#include "virus.h"

int catch_or_not ( double tprob, int npeople, int status[] ) {

  // Insert function here!

}

int die_or_not ( double dprob, int sick_days, int person, int npeople, int status[] ) {

  // Insert function here!

}

int main ( int argc, char *argv[] ) {
  const int npeople=1e+4;
  int status[npeople];
  int newstatus[npeople];
  int day,person;
  double vprob; // Vaccination probability.
  double tprob; // Transmission probability
  double dprob = 0.015; // Death probability per day
  int ndays = 100;
  int initial_infections = 0.01*npeople;
  int sick_days = 14;
  FILE *output;

  // Insert the rest of the program here!

  printf ( "Population: %d\n", npeople );
  printf ( "Vaccination Probability: %lf\n", vprob );
  printf ( "Transmission Probability: %lf\n", tprob );
  printf ( "Initial Infections: %d\n", initial_infections );
  printf ( "Simulation Period: %d days\n", ndays );
  printf ( "Number of Recovered: %d\n", nrecovered );
  printf ( "Number of Dead: %d\n", ndead );
  printf ( "Case Fatality Rate: %lf\n", (double)ndead/(nrecovered+ndead) );
}
```

tributed between zero and one (see Figure A.49). Use the following statement to generate a random number of people met that day:

```
nexposures = 20*rand01();
```

2. Then, you'll need a "`for`" statement that loops through `nexposures` people. Each time around the loop you'll need to do a couple of things:
   (a) pick a random other person from the list of people, like this:

   ```
   otherperson = rand01() * npeople
   ```

   (b) Use an "`if`" statement to see if the other person is infected (in other words, if `status[otherperson]` is greater than zero). If the other person is infected, check to see if the person we're tracking catches the disease.

   We do this by using the `rand01` function to give us a random number between zero and one, and then checking to see if that number is less than `tprob` (the "transmission probability"). If it is, the `catch_or_not` function should return a value of 1, meaning that the person is now on his first day of infection.

   If none of the encounters result in infection, `catch_or_not` should return a value of `is_susceptible`, meaning that the person isn't infected, but is still susceptible[39].

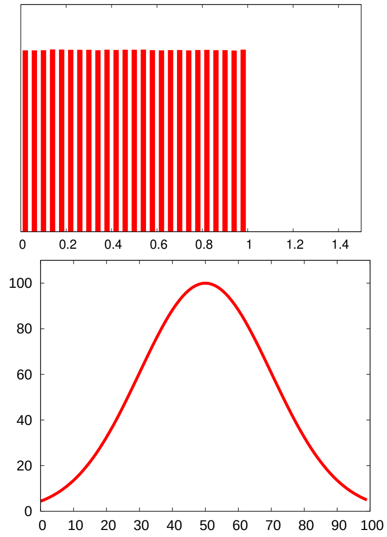


Figure A.49: Histograms of random numbers in a uniform distribution (top) from `rand01()` and a normal distribution (bottom) from `normal()`.

Writing `die_or_not`:

This function checks to see if an infected person stays sick, dies, or gets better. It takes five arguments: `dprob`, the probability of dying each day; `sick_days`, the typical number of days someone is sick[40]; `person`, this person's index in the `status` array; `npeople`, the total number of people; and the `status` array itself. Here's how to write this function:

1. Start by using the `rand01` function to give a number between zero and one, and then checking to see if that number is less than `dprob`. If it is, that means that the person died, so the function should return a value of `is_dead`.
2. If the person doesn't die, we need to check to see if she recovers or remains sick. To do this, check to see if the number of days the person has been sick is greater than `sick_days`, plus or minus a small random amount (everyone isn't sick for exactly the same amount of time). Do that like this:

   ```
       if ( status[person] > sick_days + 3.0*normal() ) {
   ```

   This uses the `normal` function, which generates random numbers that tend to be around zero, but are sometimes bigger or smaller[41]. (Remember that `status[person]` is the number of days an in-

[39] See the values in Table A.1.

[40] Notice that `simulate.cpp` already defines `dprob=0.015` and `sick_days=14`. You don't need to change these.

[41] See the bottom graph in Figure A.49.

fected person has been sick.) If this condition is true, the function should return `is_recovered`. Otherwise it should return `status[person]+1`, indicating that the person will stay infected for another day.

Writing `main`:
To complete the program, you'll need to add some lines to `main` to do the following:

1. Check to make sure the user has supplied enough command-line arguments[42]. If there aren't enough command-line arguments, the program should print out a friendly usage message and then stop without trying to do anything else[43].

2. Convert the command-line arguments into the variables `vprob` and `tprob` by using the **`atof`** function. The last command-line argument (the output file name) doesn't need to be converted. You can just use it directly, like this:

   ```
   output = fopen( argv[3], "w" );
   ```

3. Next, calculate the number of vaccinated people, using `vprob` like this:

   ```
     nvaccinated = vprob*npeople;
   ```

4. Now your program is ready to set the initial status of each person. The `virus.h` file contains a function named `initialize_status` to help you do that[44]. Use it like this:

   ```
     initialize_status( npeople, initial_infections, nvaccinated, status );
   ```

5. Your program will need a pair of nested "`for`" loops: An outer loop that goes through all the days, and an inner loop that goes through all of the individuals in the population and, for each one, checks to see whether that person gets sick, gets well, dies, or stays the same.

   The outer loop might start like this:

   ```
   for ( day=0; day<ndays; day++ ) {
   ```

   and the inner loop might start like this:

   ```
   for ( person=0; person<npeople; person++ ) {
   ```

6. Inside the inner loop we'll determine the person's new status for

[42] Syntax should be:
`./simulate vprob tprob file`

[43] See Section 9.16 of Chapter 9 for an example of how to do this.

[44] Notice that `simulate.cpp` already defines `initial_infections` to be `0.01*npeople`. You don't need to change this.

the next day (newstatus[person]). We'll need an "if" statement that has three branches that do different things depending on the person's current status:

- If the person is susceptible (in other words, if status[person] == is_susceptible) then:

  ```
  newstatus[person] = catch_or_not( tprob, npeople, status );
  ```
- If the person is infected (that is, if status[person] > 0) then:

  ```
  newstatus[person] = die_or_not( dprob, sick_days, person, npeople, status );
  ```
- Otherwise, we'll assume the person just stays the same the next day, and set newstatus[person] = status[person].

7. At the end of each day, the program should update the status of all the people by copying all of the elements of newstatus to status. The virus.h header file includes a function named update_status that will do that for us. Add the following line to your program to use it:

   ```
   update_status( npeople, status, newstatus );
   ```

8. The update_status function will also count how many people are in each state and put those numbers into the variables nsusceptible, nrecovered, nvaccinated, ninfected, and ndead. Use this fprintf line to write the those values and the current day number into the output file at the end of each day:

   ```
   fprintf ( output, "%d %d %d %d %d %d\n",
             day, nsusceptible, nrecovered, nvaccinated, ninfected, ndead );
   ```

After you've completed your program, compile it and run it like this:

```
./simulate 0.0 0.015 simulate-0.0-0.015.dat
```

That will run the simulation with vprob equal to zero (nobody vaccinated) and tprob equal to 0.015 (1.5% chance of catching the disease from an infected person). The program should print something like Figure A.50 on the screen when it's done.

```
Population: 10000
Vaccination Probability: 0.000000
Transmission Probability: 0.015000
Initial Infections: 100
Simulation Period: 100 days
Number of Recovered: 5271
Number of Dead: 1084
Case Fatality Rate: 0.170574
```

Figure A.50: Typical output from simulate.cpp with vprob=0.0 and tprob=0.015.

As you can see, under these conditions we might expect over a thousand people to die during this 100-day period. The program also prints a "Case Fatality Rate" (CFR) that tells us the likelihood of dying if you catch the disease. The CFR for this disease is about 17%. For comparison, the reported[45] global CFR for COVID-19 is about 2% and for seasonal flu in the US it's about 0.2%.

[45] See ourworldindata.org for some information about the difficulty of interpreting real-world CFR numbers.

You could graph the data in the output file using this *gnuplot* command:

```
plot "simulate-0.0-0.015.dat" using 1:2 with lines lw 5 title "Susceptible", \
 "" using 1:3 with lines lw 5 title "Recovered", \
 "" using 1:4 with lines lw 5 title "Vaccinated", \
 "" using 1:5 with lines lw 5 title "Infected",\
 "" using 1:6 with lines lw 5 title "Dead"
```

If you did, you should see a graph like the following:

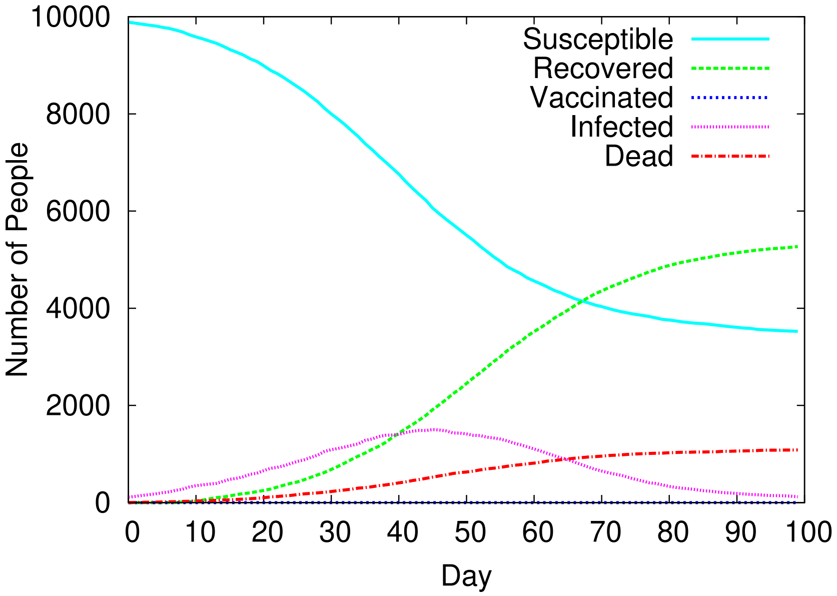


Figure A.51: Results of a simulation with `vprob=0` and `tprob=0.015`.

vprob	tprob	Deaths
0.0	0.015	1205
0.1	0.015	937
0.6	0.015	38
0.0	0.007	71
0.1	0.007	44
0.6	0.007	21

Figure A.52: Some typical values for number of deaths, given various values for `vprob` and `tprob`. (Your program's results will vary slightly from these because it uses random numbers.)

Try running the program with `vprob` set to 0.6 (60% of people vaccinated). You should see that the number of deaths is a lot smaller. Then try running it with `vprob=0.6` and `tprob=0.007` (reducing `tprob` by about half from the previous value). This simulates the effect of masking and social distancing, which reduce the transmission probability[46], and should further reduce the number of deaths. Some typical results are shown in Figure A.52.

Figures A.53 and A.54 on the next page show how you might expect the number of deaths to change as you change `vprob` and `tprob`. The graphs show that vaccinating even half the people would have a dramatic effect, and similar results could be obtained by cutting the transmission probability in half.

[46] See "Why Masks Work BETTER Than You'd Think": https://www.youtube.com/watch?v=Y47t9qLc9I4.

Program 2: How Much Do Results Vary?

When we run our simulation program it gives us numbers that tell us how many people were infected, recovered, died, and so forth, but the program works by simulating random interactions between people. It's possible that the program could sometimes just "roll the dice" in a really unlikely way, and give us results that are unrealistic.

How much random variation is there in the results our program produces? One way to get a handle on this would be by running the program many times with the same set of parameters, and then looking at how much the results vary.

We don't have to do that by hand, though. As we've seen, computers are very good at doing the same thing over and over again very quickly. We can just get the computer to run our simulation many times for us. That's what you'll do for your second program, which will be called **analyze.cpp**.

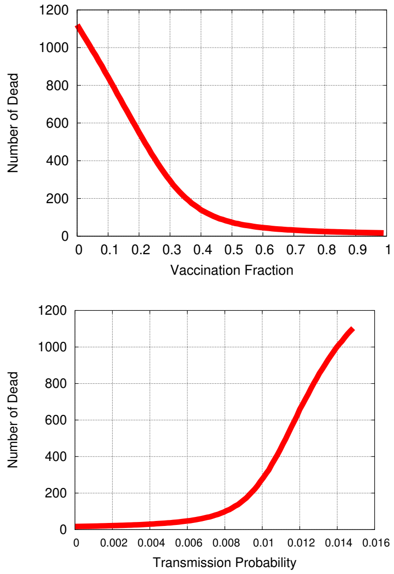


Figure A.53: The top graph shows the number of deaths after 100 days for various values of `vprob` when `tprob` is 1.5%. The bottom graph shows number of deaths for various values of `tprob` when `vprob` is 0%.

How the Program Works

This new program will start out as a copy of `simulate.cpp`. Instead of just simulating 100 days once, though, the new program will do the same simulation 1,000 times. At the end, it will tell the user the average number of deaths after 100 days, and the standard deviation of this number. The program will also write the results of each trial into an output file for later analysis. (We'll use this file with the third program you write.)

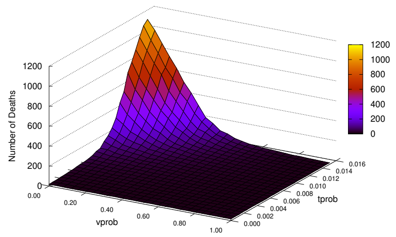


Figure A.54: Effect of changing `vprob` and `tprob` on the number of deaths after 100 days.

How to Write the Program

Follow these steps to write `analyze.cpp`:

1. Start by copying `simulate.cpp`:

   ```
   cp simulate.cpp analyze.cpp
   ```
2. Then edit `analyze.cpp` and add a few new variables to `main`:

   ```
     double sum=0;
     double sum2=0;
     int trial;
     int ntrials=1000;
   ```

   We'll use these for looping through 1,000 trials and calculating the average and standard deviation of the number of deaths.

3. Next, you'll need to add a "`for`" loop around most of the rest of `main`. The "`for`" loop should start before the `initialize_status` line (so that the status of all the people gets reset at the beginning of each trial), and it should begin like this:

```
for ( trial=0; trial<ntrials; trial++ ) {
```

4. This program will take a while to run, so immediately under the beginning of the new "`for`" loop you should include some lines to print progress messages. Do this by printing a message like "Processing trial 20" whenever `trial` is a multiple of ten[47].

5. Remove the existing `fprintf` statement, since we don't want this program to write results at the end of every day. (We'll add a new `fprintf` statement in an other place – see below – to write the results at the end of each trial.)

6. At the end of each trial, do a couple of things:

   - Add `ndead` to `sum`, and add `ndead*ndead` to `sum2`. We'll use these sums later to calculate the mean and standard deviation after we've done all the trials.
   - Write the results from this trial into the output file. Use a line like this:

```
fprintf ( output, "%d %d %d %d %d %d\n",
          trial, nsusceptible, nrecovered, nvaccinated, ninfected, ndead );
```

7. At the very end of the program, instead of printing the "Number of Recovered", "Number of dead", and "Case Fatality Rate" instead print:

   (a) The number of trials
   (b) The average number of deaths at the end of each trial
   (c) The standard deviation of this number

Compile `analyze.cpp` and try running it like this:

```
./analyze 0.0 0.015 analyze-0.0-0.015.dat
```

It should tell you something similar to Figure A.55, showing that, if nobody is vaccinated and people have a 1.5% chance of catching the virus from an infected person, the expected number of deaths after 100 days is about 1117 ±34, so between 1083 and 1151.

```
Population: 10000
Vaccination Probability: 0.000000
Transmission Probability: 0.015000
Initial Infections: 100
Simulation Period: 100 days
Number of Trials: 1000
Average Number of Dead: 1117.258000
Standard Deviation: 34.126149
```

Figure A.55: Typical output from `analyze.cpp` when `vprob=0.0` and `tprob=0.015`.

If you set `vprob=0.6` and `tprob=0.007` you should see that the

[47] See Section 4.4 of Chapter 4 for information about how to do this.

number of deaths after 100 days drops to about 20 ±5.

Program 3: Visualizing the Variation

While `analyze.cpp` is running it writes the results of each trial into an output file. Your next job is to write a new program named **`visualize.cpp`** that will read that file and produce a histogram. Your program will be similar to program 7.1 in Chapter 7. The data it will histogram is the number of deaths. It should start out like Program A.19 below. Notice that it defines a 50-element array, `bin`, to hold the histogram data.

Lambs, Nantucket (1874), by Eastman Johnson, National Gallery of Art.
Source: *Wikimedia Commons*

Program A.19: visualize.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main ( int argc, char *argv[] ) {
  const int nbins=50;
  int bin[nbins];
  double dmin, dmax;
  double binsize;
  int trial;
  int nsusceptible, nrecovered, nvaccinated, ninfected, ndead;
  int binno;
  int overunderflow=0;
  int i;
  FILE *input;
  FILE *output;

  // Insert Program Here.
}
```

How the Program Works

Like the preceding programs, this one will expect parameters on its command line, and should complain and exit if it doesn't get the proper number of parameters. Its usage will be:

```
./visualize dmin dmax inputfile outputfile
```

where `dmin` and `dmax` are the minimum and maximum number of deaths that can be recorded in the histogram; `inputfile` is the name of a file that was produced by `analyze.cpp`; and `outputfile` is the name of a file into which the new program will write the histogram data. When plotted, the data should look like Figure A.56 on page 580.

How to Write the Program

To make the histogram, the program should proceed as follows:

1. Check the number of command-line arguments, and use `atof` to

set the values of dmin and dmax. The input and output files can be opened like this[48]:

```
input  = fopen(argv[3],"r");
output = fopen(argv[4],"w");
```

2. Then, determine the binsize, like this:

```
binsize = (dmax-dmin)/nbins;
```

3. Next, use a while loop to read data from the input file[49]. Each line of the file will contain six integer values: trial, nsusceptible, nrecovered, nvaccinated, ninfected, and ndead.

4. Determine which bin this ndead value belongs in, and increment that bin. Be sure to keep a count of the number of over/underflows, as Program 7.1 does. Since the range of our histogram is dmin to dmax, the bin number will be:

```
binno = (ndead-dmin)/binsize;
```

5. After processing all of the input data, write the histogram data into the output file. For each bin of the histogram, write two numbers separated by a single space: the ndead value represented by that bin[50], and the value of bin[i]. The ndead value can be calculated from the bin number, like this:

```
ndead = dmin + binsize*(0.5+i);
```

   where i is the bin number.

6. Finally, at the bottom of the output file, write a line beginning with a # that tells how many overflows or underflows were seen.

Compile your program and try running it like this, using the output file you created above[51]:

```
./visualize 1000 1250 analyze-0.0-0.015.dat visualize-0.0-0.015.dat
```

You can plot the resulting data file with *gnuplot* like this:

```
plot "visualize-0.0-0.015.dat" with impulses lw 5
```

The result should look like Figure A.56, which seems to be in agreement with our previous program's prediction that the number of deaths with vprob=0 and tprob=0.015 would typically lie somewhere between 1083 and 1151.

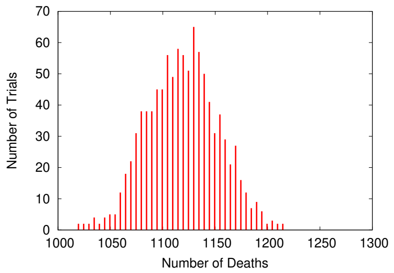


Figure A.56: Histogram of number of deaths after 100 days, from 1,000 trials with vprob=0 and tprob=0.015.

[48] Notice that we open one file for reading (with "r") and the other for writing (with "w").

[49] See Chapter 5 for information about reading data from files.

[50] Note that this is different from Program 7.1, where we just printed the bin number as the first column in the output file.

[51] Note that I've chosen values of dmin and dmax that are separated by a multiple of 50 (the number of bins). Doing this will make your graphs look better.

Conclusion

Congratulations! The programs you've written will help you and your boss know what to expect if this disease reaches Nantucket. It appears that if you can get 60% of the people vaccinated and use mask-wearing and social distancing to reduce transmission by half, the number of deaths can be reduced from over a thousand to only twenty or so. This is still tragic, but the lives of a thousand people could be saved.

An inflatable sea serpent on a Nantucket beach (1937).
Source: Wikimedia Commons

The programs you've written are very simple compared to the sophisticated models that organizations like the CDC use for forecasting the spread of a disease, but they still give you some insight in to how a pandemic works and a couple of the factors that influence its progress. You might think about how you could improve on the programs you've written. Here are some things to consider:

The first surfboard on Nantucket (1932).
Source: Wikimedia Commons

- Rather than assuming a perfect vaccine, what if the vaccine only reduced your probability of getting infected? What if it reduced your probability of dying?
- What if the simulation allowed you to put a limit on the size of gatherings by changing how `nexposures` is calculated?
- What if infected people were quarantined after they showed symptoms?
- What if the simulation increased the probability of death after the number of sick people exceeded the number of available hospital beds?
- What if conditions change during the course of the outbreak? `tprob` might be higher in cold weather, for example, or people might be getting vaccinated while the disease is spreading.
- What about the effect of social networks? People tend to interact with a network of friends, relatives, and co-workers rather than people chosen completely at random.

Because of the increasing global population and the growth of global travel and commerce, forecasting disease outbreaks is more important than ever before. Sara Del Valle, a mathematical epidemiologist at Los Alamos National Laboratory, argues that we need to make disease forecasting as accurate and reliable as weather forecasting[52]. As a step in this direction, in 2021 the CDC announced the formation of a new "Center for Forecasting and Outbreak Analytics"[53]. Efforts like this should provide fertile new ground for young programmers who want to make the world a better place.

[52] Sara Del Valle, "We need to forecast epidemics like we forecast the weather".

[53] CDC press release

Header Files

You'll need the following two header files in order to write your programs.

The first file, `random.h`, contains two functions for generating random numbers. See Figure A.49 on page 573 for an illustration of the output from the two functions.

Program A.20: random.h

```
// Random number between zero and one
double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }
  return ( rand()/(1.0+RAND_MAX) );
}
// Normal distribution:
double normal () {
  int nroll = 12;
  double sum = 0;
  int i;
  for ( i=0; i<nroll; i++ ) {
    sum += rand01();
  }
  return ( sum - 6.0 );
}
```

The second file, `virus.h`, contains some global variable definitions and two functions: `update_status` and `initialize_status`.

The `initialize_status` function takes care of setting up the initial health status of all 10,000 people. Depending on the values of `initial_infections` and `nvaccinated`, some number of people are set to the infected or vaccinated states. Everybody else is set to the susceptible state.

The `update_status` function takes care of copying information from `newstatus` to `status` and also counts how many people have each health status.

Program A.21: virus.h

```
// Global variables:

// Counters:
int nsusceptible, ninfected, nrecovered, nvaccinated, ndead;

// Status indicators:
// Any positive value indicates user has been infected for n days.
// Negative values indicate:
const int is_susceptible = 0;
const int is_recovered = -1;
const int is_vaccinated = -2;
const int is_dead = -3;

// Update status of each person and count them
void update_status( int npeople, int status[], int newstatus[] ) {
  int i;

  // Reset counters in preparation for counting:
  nsusceptible = 0;
  ninfected = 0;
  nrecovered = 0;
  ndead = 0;

  for ( i=0; i<npeople; i++ ) {
    status[i] = newstatus[i];
    if ( status[i] == is_susceptible ) {
      nsusceptible++;
    } else if ( status[i] == is_recovered ) {
      nrecovered++;
    } else if ( status[i] == is_dead ) {
      ndead++;
    } else if ( status[i] > 0 ) {
      ninfected++;
    }
  }
}
void initialize_status ( int npeople,
                         int initial_infections, int nvaccinated, int status[] ) {
  int i;
  int vmax;

  // Initial infections:
  for ( i=0; i<initial_infections; i++ ) {
    status[i] = 1;
  }

  // Vaccinations:
  // Don't exceed total number of people!
  vmax = initial_infections+nvaccinated;
  if ( vmax > npeople ) {
    vmax = npeople;
  }
  for ( i=initial_infections; i<vmax; i++ ) {
    status[i] = is_vaccinated;
  }

  // Everybody else is susceptible:
  for ( i=vmax; i<npeople; i++ ) {
    status[i] = is_susceptible;
  }

}
```

Project 7: Crowd Control

Introduction

Imagine you're in charge of security for a stadium. The stadium hosts large sports events and concerts. It holds 10,000 people. Your job is to move these people safely into the stadium during the 90 minutes before an event begins.

To enter the stadium, attendees need to pass through a security screening process that includes bag checks and metal detectors. To help get people into the stadium quickly, there are 20 different security check stations distributed around the stadium's perimeter.

In this project we'll be writing three programs that simulate the process of passing spectators into such a stadium, and analyze and visualize the results of our simulation. With our programs we'll try to answer questions that would come up in the real world:

- How long does it take to get everybody into the stadium?
- How long do people wait in line?
- How long do the lines get?

Our simulation will have two adjustable parameters: The rate at which people arrive, and the average time needed to do a security scan on a person. You'll be writing three programs, named `simulate.cpp`, `analyze.cpp`, and `visualize.cpp`.

The Roman Colosseum (top photo) could hold 50,000 to 80,000 spectators. The University of Virginia's Scott Stadium (middle photo) accommodates 60,000. Currently, the largest stadium in the world is Narendra Modi Stadium (bottom photo) in India, which holds 132,000 people.
Source: Wikimedia Commons (top), University of Virginia (middle), Gujarat Cricket Association (bottom)

Program 1: Simulating the Lines

Your first program will be called `simulate.cpp`. It will simulate the process of moving 10,000 people into a stadium, using 20 entrance lines. To answer the questions listed in the introduction we'll need to keep track of a few things for each line:

- When did a person join the line?
- Who's next in line?
- How many people are currently in line?

In programming, we often refer to a line like this as a "queue", and I'll use those words interchangeably in the following descriptions.

Our program will keep track of the necessary information by using arrays. To start with, imagine that we only had one queue leading to one scanning station. We could have an array that stored the time at which each person entered the line, as shown in Figure A.57.

In the top diagram we see that there are six people in line. The first person (element 0 of the array) entered the line 49 seconds after the gates opened. The other people entered at successively later times. A variable named `qstart` keeps track of who is at the front of the line (in this case, person 0). The variable `qend` keeps track of the next free spot in line (element number 6 of the array, in this case). The number of people in line at this point is just `qend - qstart`.

When we want to add a person to the line, we put the person's arrival time into element `qend`, and then add one to the value of `qend`, as shown in the middle diagram. There are now 7 people in line, which is equal to the new value of `qend - qstart`.

When the person at the front of the line gets passed through the scanning station, we add 1 to `qstart`, as shown in the bottom diagram. Now `qend - qstart` is 6 again, the number of people still in line.

If we have more than one line, we'll need to keep track of `qstart` and `qend` for each of them. We can do this by making `qstart` and `qend` arrays. If we have 20 lines, numbered 0 through 20, we could have `qstart[0]` be the value of `qstart` for line number 0, `qstart[1]` for line number 1, and so forth.

For keeping track of everybody's arrival time, let's use a 2-dimensional array named `q`. In our program, we'll define `q`, `qstart`, and `qend` like this[54]:

```
int q[ nqueue ][ npeople ];
int qstart[ nqueue ] = {0};
int qend[ nqueue ] = {0};
```

where `nqueue` is the number of queues ("lines"), and `npeople` is the number of people in our simulation. `npeople` is the maximum possible number of people who can be standing in line, even if we

In 2005, UVa's Scott Stadium hosted a Rolling Stones concert.
Source: Wikimedia Commons

Bono and Adam Clayton during a U2 concert at Scott Stadium on October 1, 2009.
Source: Wikimedia Commons

In 2017, Dave Matthews and many other musicians played a "Concert for Charlottesville" in Scott Stadium, in response to the violent events in August of that year.
Source: Wikimedia Commons

[54] Notice that we set all the values of `qstart` and `qend` to zero initially.

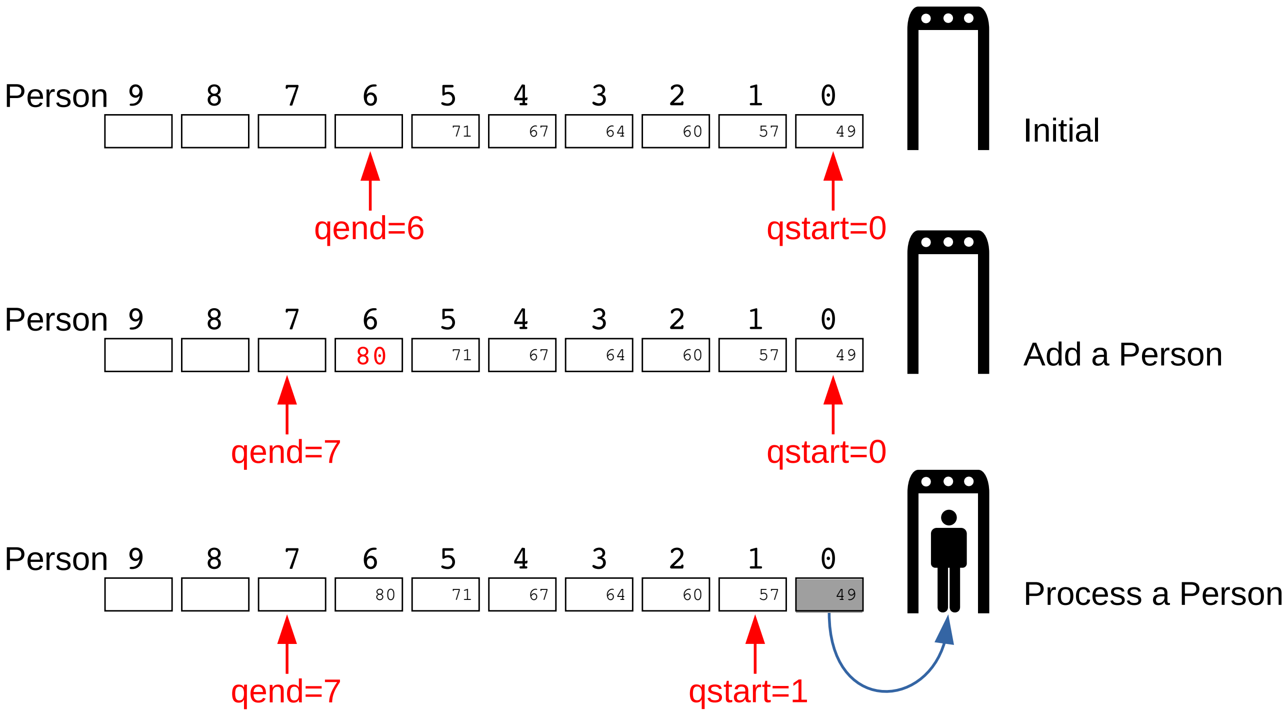


Figure A.57: This diagram shows how our program will add and remove people from a queue.

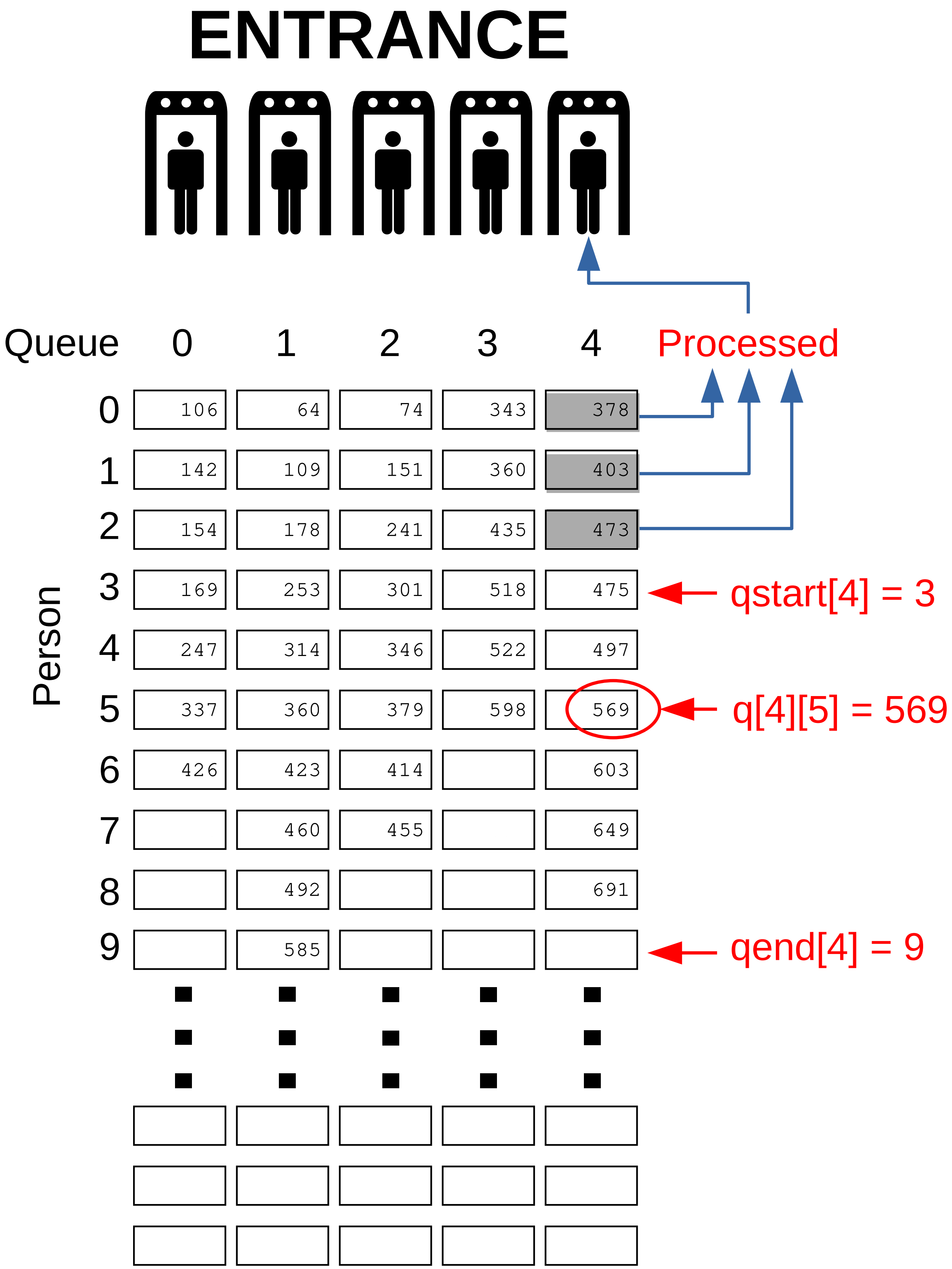
only had one line. Figure A.58 shows what the situation will now look like with multiple queues[55]. Here we show a situation in which the first three people in queue 4 have been processed through the scanning station. The number of people left in queue 4 is `qend[4]-qstart[4]` = 6. If we wanted to know when person number 5 entered this queue, we could look at `q[4][5]` and see that he arrived at 569 seconds after the gates opened.

How to Write the Program

Writing the simulation program will take a little work, but fortunately you have an assistant who's already done some of it for you. Program A.22 shows what he's written so far. You'll just need to fill in the missing pieces (marked with "ADD").

Notice that the program has a variable named `t` that keeps track of the elapsed time. It starts out with `t = 0`. At the bottom of the program's `do-while` loop, 1 second is added to `t` every time the program goes around the loop. Each trip around the loop represents the passage of 1 second of time. During that time, newly-arrived people can join the queues, and people at the front of the queues can be scanned and passed into the stadium.

[55] For clarity, this only shows five of the twenty queues.



Person \ Queue	0	1	2	3	4
0	106	64	74	343	378
1	142	109	151	360	403
2	154	178	241	435	473
3	169	253	301	518	475
4	247	314	346	522	497
5	337	360	379	598	569
6	426	423	414		603
7		460	455		649
8		492			691
9		585			

Figure A.58: A diagram illustrating our program's strategy for simulating the queues.

Program A.22: simulate.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main ( int argc, char *argv[] ) {
  const int npeople = 10000;
  const int nqueue = 20;
  int q[nqueue][npeople];
  int qstart[nqueue] = {0};
  int qend[nqueue] = {0};
  int currentq = 0;
  int i;
  int t = 0;
  int nscanned = 0;
  int nqueued = 0;
  int max_arrivals_per_second;
  int arrivals;
  double scan_delay;
  double flip;
  FILE *output;

  // ADD: Check number of command-line arguments.
  // ADD: Set max_arrivals_per_second and scan_delay.
  // ADD: Open output file

  srand(time(NULL));

  do {

    // ADD: Find number of new arrivals:
    // ADD: Add new arrivals to the back of the lines:

    // Process people at the front of the lines:
    for ( i=0; i<nqueue; i++ ) {
      if ( qend[i] - qstart[i] > 0 ) { // Is there anyone in the queue?
        flip = rand()/(1.0+RAND_MAX);
        if ( flip < 1.0/scan_delay ) { // Is the scanner ready for a new person?

          // ADD: Write data to output file.
          // ADD: Update qstart.
          // ADD: Count number of people scanned so far.

        }
      }
    }

    t++;
  } while ( nscanned != npeople );

  printf( "Elapsed time %d seconds (%lf hours)\n", t, t/3600.0);

  fclose( output );
}
```

The program uses the variable nqueued to keep track of how many people have arrived so far, and it uses nscanned to keep track of how many people have been passed into the stadium. The variable npeople is set to 10,000, the total number of attendees.

The program should be written so that you can run it with command-line arguments like this:

```
./simulate 5 10 simulate-5-10.dat
```

where the first number sets the value of the variable max_arrivals_per_second. This will control the rate at which people arrive. The second number sets scan_delay[56]. This controls the average time (in seconds) required to do a security scan on a person. The last thing on the command line is the name of a file into which the program will write its output.

Here's a list of the things you'll need to add to the program:

1. Check to make sure the user has supplied enough command-line arguments. If there aren't enough command-line arguments, the program should print a friendly usage message and then stop without trying to do anything else[57].

2. Convert the command-line arguments into the variables max_arrivals_per_second and scan_delay by using the atoi and atof functions[58]. The last command-line argument (the output file name) doesn't need to be converted. You can just use it directly, like this:

   ```
   output = fopen( argv[3], "w" );
   ```

3. Inside the do-while loop you'll first need to determine how many new arrivals there are during this second. We'll add these people to the total number of people who have arrived (nqueued) and put their arrival time into slots in the queues.

   When you run the program you'll specify max_arrivals_per_second on the command line. We'll assume that the number of new arrivals during each trip around the loop will be some random number less than this. We can start by saying:

   ```
   arrivals = max_arrivals_per_second*(rand()/(1.0+RAND_MAX));
   ```

   but we need to make sure that nqueued never gets bigger than npeople (the total number of attendees). To prevent that, we need

[56] Notice that max_arrivals_per_second and scan_delay are defined as an int and a double, respectively, near the top of Program A.22.

[57] See Section 9.16 of Chapter 9 for an example of how to do this.

[58] Notice that max_arrivals_per_second is an int and scan_delay is a double.

to add something like this next:

```
if ( nqueued + arrivals > npeople ) {
  arrivals = npeople - nqueued;
}
```

Now `arrivals` will be zero when `nqueued` gets as big as `npeople`.

A happy crowd at a football game in Scott Stadium.

4. Now that we know how many people arrived, we need to add these people, one by one, to the queues. To do that we'll need a "`for`" loop that starts like "`for (i=0;i<arrivals;i++)`". Inside the loop we'll need to do a few things.

   We'll need to decide which queue we'll put this person into. There are several different reasonable ways to do this, but let's use something called "round-robin" queueing. In that way of doing it, we add the first person to queue 0, the second person to queue 1, and so on until we get to the last queue, and then we start back at queue 0 again. To keep track of which queue is the current one, we'll use the variable `currentq` which initially has the value 0.

   We'll also need to know where to put this person into the current queue. The first open slot of this queue is given by `qend[ currentq ]`. With all that in mind, here's what we need to do for each person who's just arrived:

A game in progress (UVa versus Wake Forest, 2007).
Source: Wikimedia Commons

   (a) Add each of the new arrivals to the current queue by putting the current time (`t`) into `q[ currentq ][ qend[currentq] ]`.
   (b) Then we need to add 1 to `qend[currentq]`, so that it points to the next unfilled slot (See Figure A.57.)
   (c) Next we need to figure out which queue we'll be putting the next person into. We can do that like this:

   ```
   currentq = (currentq+1)%nqueue;
   ```

   What we're doing here is adding 1 to `currentq`, but making sure that we go back to zero after the last queue. The `%` is the modulo operator, which gives the remainder after division. It makes our calculation work like the numbers on a clock face: When we've gone through all of the numbers, we start back at zero.
   (d) Finally, we need to add 1 to `nqueued` to indicate that one more person has successfully arrived.

Scott Stadium covered in snow.

5. Now that we've added any new arrivals, we need to take care of the people at the front of the lines. The number of queues

is given by `nqueue` (this is set to 20 at the top of the program). We need to process the people standing at the front of each of these queues, so we'll need a "`for`" loop that starts out like this: "`for (i=0;i<nqueue;i++)`". Inside this loop we'll need to do the following for each queue[59]:

(a) First, let's see if anyone is waiting in this line. (Maybe the line is empty right now!) The number of people waiting in one of the queues is `qend[i] - qstart[i]`. If this is greater than zero, then someone is waiting to get into the stadium, and we should go on to the next steps.

(b) Next, we'll need to see if the gate attendant has finished scanning this person. We'll decide by "flipping a coin". We do this by generating a random number between zero and one[60]. The program should put this number into the variable named "`flip`", for use in the next step.

(c) The average number of seconds it takes for a security scan is given by `scan_delay`, so we're going to assume that the probability of finishing a scan during one of our 1-second-long trips through the `do-while` loop is `1.0/scan_delay`. To decide whether this person is ready to pass into the stadium, check to see if `flip` is less than `1.0/scan_delay`. If it is, then proceed to the next step.

(d) At this point, we know that someone is waiting in line and that the gate attendant is ready to pass them into the stadium. We just have three small things to do in order to finish processing this person:

   i. Print the information about this person into the output file. For each person processed we want to print the following 4 things[61]:

   - The queue number, `i`,
   - The current time, `t`,
   - The amount of time the person has been waiting, `t-q[i][qstart[i]]`,
   - The number of people currently in this line, `qend[i]-qstart[i]`

   Each of those things is an integer. They should be printed into the output file in the order above, with spaces between them and with a "`\n`" at the end of the line.

   ii. We need to add 1 to `qstart[i]` (See the bottom of Figure A.57).

   iii. The variable `nscanned` will keep track of how many people have passed into the stadium. It's initially set to zero at the top of the program. Now we need to add 1 to it, to indicate that we've finished processing another person. Our program's `do-while` loop will stop when `nscanned` is equal to `npeople`.

[59] Steps *a* through *c* have already been written by your assistant. You'll need to add the things in step *d*.

[60] See Section 2.6 of Chapter 2 for an example of this.

[61] See Program 5.3 in Chapter 5 for an example of writing numbers into a file.

Running the Program

After you've finished writing and compiling the program try running it a few times, like this:

```
./simulate 5 5 simulate-5-5.dat
./simulate 5 10 simulate-5-10.dat
./simulate 5 30 simulate-5-30.dat
```

Arrivals/sec	Delay (sec)	Time (hours)
5	5	1.36
5	10	1.49
5	30	4.43
30	5	0.76
30	10	1.51
30	30	4.54

Figure A.59: This table shows the total amount of time it would take to get all 10,000 people into the stadium, given a few different values of `max_arrivals_per_second` and `scan_delay`.

Those commands tell the program to simulate 5 people/sec arriving, with a scan delay of 5, 10, and 30 seconds. Notice the total elapsed time that the program prints in each case. Figure A.59 shows the time required to move everybody into the stadium for a few different choices of `max_arrivals_per_second` and `scan_delay`.

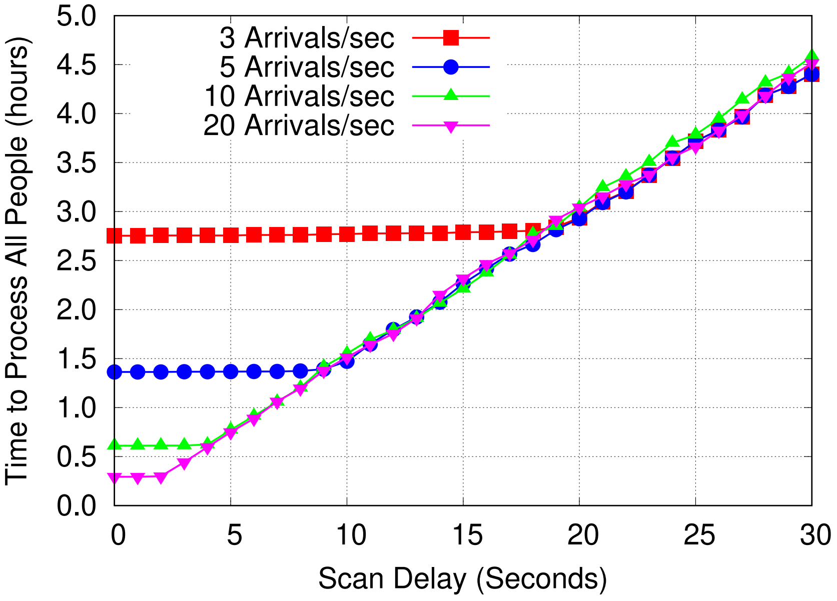


Figure A.60: The graph at left shows the time required to get all the people into the stadium as a function of `scan_delay`, for several different values of `max_arrivals_per_second`.

If you did this for many different values of these two parameters you could make a graph like Figure A.60. The vertical axis shows how long it takes to get all the people into the stadium, and the horizontal axis shows `scan_delay`. Different kinds of lines show different values of `max_arrivals_per_second`.

Clearly, changing the `scan_delay` can make a big difference in the way the queues behave. For any given value of `max_arrivals_per_second` there seems to be a critical value of `scan_delay`, beyond which it starts taking longer and longer to get everybody into the stadium. For 5 arrivals/second, this critical delay is about 10 seconds.

We could use graphs like Figure A.60 or those shown in Figure A.61 to examine our data, but it's useful to have a quantitative summary of our results. That's what we'll do in our second program.

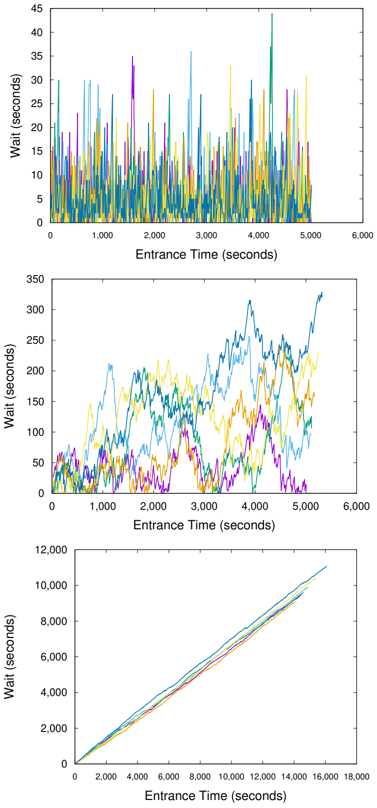


Figure A.61: These figures show how long people waited in line, versus when they finally entered the stadium. Different colors represent different queues (six queues are shown here). In all figures, the `max_arrivals_per_second` was set to 5. For the top figure, the `scan_delay` was 5 seconds. For the middle figure, it was 10 seconds. For the bottom figure, it was 30 seconds.

Program 2: Analyzing the Simulation Data

Your second program will be named `analyze.cpp`. It will read a data file produced by your first program, and give you a statistical summary of the data it contains.

Like the first program, `analyze.cpp` should accept all of its parameters on the command line, and give users a helpful message if they don't give it the right number of arguments. The usage should be:

```
./analyze filename
```

where `filename` is the name of a data file produced by your `simulate.cpp` program.

Once again, your assistant has already started writing the program for you. You can see his work in Program A.23.

Program A.23: analyze.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
int main ( int argc, char *argv[] ) {
  FILE *input;
  int queue, t, wait, length;
  int maxwait, maxlength, maxtime;
  double waitsum=0, waitsum2=0, mean, stddev;
  int npeople=0;
  int initialized=0;

  // ADD: Write the rest of the program here.

  printf ("Analyzed %d people.\n", npeople );
  printf ("Maximum wait was %d seconds (%lf minutes)\n", maxwait, maxwait/60.0 );
  printf ("Average wait was %lf seconds (%lf minutes)\n", mean, mean/60.0 );
  printf ("Standard deviation was %lf seconds (%lf minutes)\n", stddev, stddev/60.0 );
  printf ("Maximum queue length was %d people\n", maxlength);
  printf ("Time to fill stadium was %d seconds (%lf hours)\n",
         maxtime, maxtime/60.0/60.0 );
}
```

How to Write the Program

Here's a list of things you'll need to add to the program:

1. As you did in the first program, check to make sure the user has supplied the correct number of command-line arguments, and exit with a friendly message if the number isn't right.

2. The only command-line argument for this program is the file name, which doesn't need to be converted in any way. It can just be used directly like this:

   ```
   input = fopen( argv[1], "r" );
   ```

3. Now you'll need a "`while`" loop to read the file you've opened[62]. Our program will read four values from each line of the data file:

   - **`queue`**, The queue from which this person entered the stadium.
   - **`t`**, The time at which the person entered the stadium.
   - **`wait`**, How long the person waited in line.
   - **`length`**, How many people were in line at that time.

   All of these values are integers, and variables to hold them are already in Program A.23.

4. Inside the loop, you'll need to do a few things:

   (a) Add up the `wait` values you read from the file. Program A.23 already has a variable named `waitsum` for this. You'll need this sum later to calculate the mean value of `wait`.

   (b) You'll also need to find the sum of the *squares* of the `wait` values. Program A.23 already has a variable named `waitsum2` for this. This sum will be used later for finding standard deviation of `wait`.

   (c) You'll also need to count the number of lines you've read from the file. You'll need this later, along with the sum and sum of squares, to calculate means and standard deviations.

   (d) Lastly, you'll need to keep track of the maximum values of `wait`, `length`, and `t`. Refer to Program 5.4 in Chapter 5 for an example of how to do this. In our analyze.cpp program, we'll need something like this for each thing we want to find the maximum of:

   ```
   if ( initialized == 0 || wait > maxwait ) {
     maxwait = wait;
   }
   ```

   so there should be three "`if`" statements like this (one for `wait`, one for `length`, and one for `t`). After all three "`if`" statements are done, we'll need to tell the program that we now have initial guesses at the maximum values, so we should put in a line that says:

   ```
   initialized = 1;
   ```

5. After the "`while`" loop is done, we just need to calculate the mean and standard deviation of `wait`. The `printf` statements that are already in Program A.23 will then print out the results.

An early picture of UVa's marching band, taken by Rufus W. Holsinger (1866-1930).
Source: Wikimedia Commons

A 1919 football game, photographed by Holsinger.

[62] See Program 7.4 in Chapter 7 for an example of this. Refer to that program to see how to read the data and calculate the average and standard deviation as you read data from the input file.)

If you use the program to analyze the file `simulate-5-10.dat` that you created earlier, you should see output like this:

```
Analyzed 10000 people.
Maximum wait was 376 seconds (6.266667 minutes)
Average wait was 104.385000 seconds (1.739750 minutes)
Standard deviation was 80.031982 seconds (1.333866 minutes)
Maximum queue length was 40 people
Time to fill stadium was 5330 seconds (1.480556 hours)
```

Program 3: Visualizing the Wait Times

Your next job is to write a new program named `visualize.cpp` that will read the files produced by `simulate.cpp` and produce a histogram of the wait times. Your program will be similar to program 7.1 in Chapter 7. Your assistant has started writing it for you, as you can see in Program A.24 below. Notice that it defines a 50-element array, `bin`, to hold the histogram data.

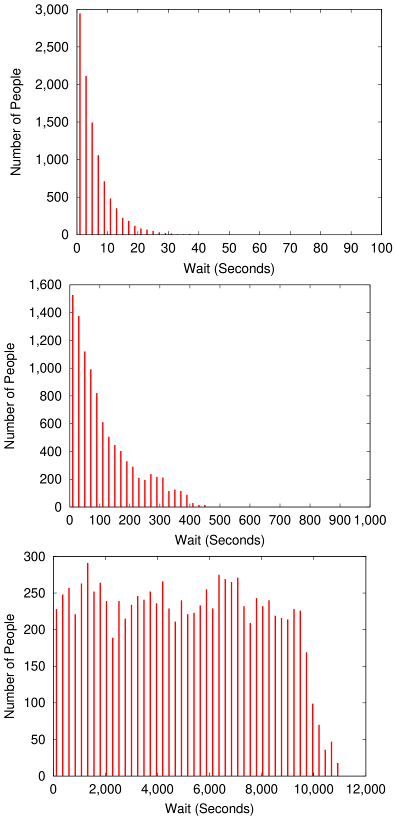


Figure A.62: Histograms of wait times for 5 arrivals/sec, and delays of 5 (top), 10 (middle), and 30 (bottom) seconds.

Program A.24: visualize.cpp

```
#include <stdio.h>
#include <stdlib.h>
int main ( int argc, char *argv[] ) {
  const int nbins=50;
  int bin[nbins] = {0};
  double min, max;
  double binsize;
  int binno;
  int overunderflow=0;
  int i;
  int t, queue, wait, length;
  FILE *input;
  FILE *output;

  // ADD: Write the rest of the program here.

}
```

How to Write the Program

Like the preceding programs, this one will expect parameters on its command line, and should complain and exit if it doesn't get the proper number of parameters. Its usage will be:

```
./visualize min max inputfile outputfile
```

where min and max are the minimum and maximum wait times that can be recorded in the histogram; inputfile is the name of a file that was produced by analyze.cpp; and outputfile is the name of a file into which the new program will write the histogram data. When plotted, the data should look like Figure A.62 on page 596.

To make the histogram, the program should proceed as follows:

1. Check the number of command-line arguments, and use atof to set the values of min and max. The input and output files can be opened like this[63]:

   ```
   input  = fopen(argv[3],"r");
   output = fopen(argv[4],"w");
   ```

2. Then, determine the binsize, like this:

   ```
   binsize = (max-min)/nbins;
   ```

3. Next, use a "while" loop to read data from the input file[64]. Your "while" loop should start out exactly like the loop in your analyze.cpp program, since it will be reading the same files.

4. Inside the loop, determine which bin each wait value belongs in, and increment that bin. Be sure to keep a count of the number of over/underflows, as Program 7.1 does. Since the range of our histogram is min to max, the bin number will be:

   ```
   binno = (wait-min)/binsize;
   ```

5. After processing all of the input data, write the histogram data into the output file. For each bin of the histogram, write two numbers separated by a single space: the wait value represented by that bin[65], and the value of bin[i]. The wait value can be calculated from the bin number, like this:

   ```
   wait = min + binsize*(0.5+i);
   ```

   where i is the bin number.

6. Finally, at the bottom of the output file, write a line beginning with a # that tells how many overflows or underflows were seen.

Compile your program and try running it like this, using one of the data files created by your simulate.cpp program:

[63] Notice that we open one file for reading (with "r") and the other for writing (with "w").

[64] See Chapter 5 for information about reading data from files.

[65] Note that this is different from Program 7.1, where we just printed the bin number as the first column in the output file.

```
./visualize 0 1000 simulate-5-10.dat visualize-5-10.dat
```

You can plot the resulting data file with *gnuplot* like this:

```
plot "visualize-5-10.dat" with impulses lw 5
```

Your results should look like the graphs in Figure A.62.

Final Thoughts

Congratulations! The simulation, analysis, and visualization programs you've written are powerful tools that will help you keep the spectators happy. Apparently, by controlling the amount of time it takes to do a security scan, and the rate at which people arrive, we can assure that everybody gets into the stadium quickly, nobody has to wait in line too long, and the lengths of the lines aren't too great.

For example, by running

```
./analyze simulate-5-10.dat
```

we can see that, if we limit the arrival rate to 5 people per second, and if our security scan takes 10 seconds on average, we can get everybody into the stadium in 1.5 hours, with people only standing in line for a couple of minutes, and never having more than about 40 people in line at a time.

It's interesting that our results show a remarkable change in the behavior of the queues when we change from a small scan delay to a large one. Take a look again at Figure A.60 and Figure A.61 on page 593.

For small delays, people flow quickly into the stadium. Sometimes there's nobody in line at all. The wait times vary up and down randomly. For long delays, the wait times are long and predictable, falling on a straight line. What we're seeing here is a transition between what we'd call "turbulent flow" and "laminar flow" in fluid dynamics. In the small-delay case there's not much "friction". The crowd behaves like a low-viscosity fluid, such as water. In the long-delay case, the flow is dominated by the "friction" due to the scanning process, and becomes slow and steady, like pouring syrup.[66]

Source: Wikimedia Commons (top), Wikimedia Commons (bottom)

[66] If we did a little math, we could show that the transition from turbulence to laminar flow happens when:

$$\texttt{scan\_delay} = \frac{2 * \texttt{nqueue}}{\texttt{max\_arrivals\_per\_second} - 1}$$

This is the point where people start piling up in the queues because they aren't being scanned fast enough.

Project 8: Auto-Compose

Introduction

Imagine you have a friend who's a songwriter. Your friend has run out of ideas for new melodies, so she asks if you can use your programming skills to give her some inspiration. Let's use a little musical mathematics to generate some random tunes.

The human mind is wired for finding patterns. We often think of visual patterns, but our ears can also find patterns in sound. For example, as shown in Figure A.63, when a sound has a frequency that's twice that of another sound, the two sounds seem harmonious to us. Our mind knows that they're somehow related.

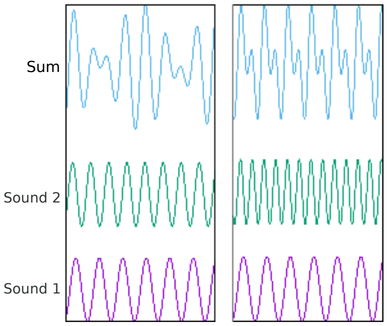


Figure A.63: If we pick two sound frequencies at random, they'll probably combine into something jangly and unpleasant, as shown on the left. If one frequency is twice the other, though, as shown on the right, the two sounds form a nice repeating pattern that's noticed by our ears.

For millenia, musicians have used this principle to organize sound into *octaves*, where an octave is a range that starts at some base frequency and goes up to twice that starting frequency. In Western music, an octave is typically divided into twelve sections, each starting with a different frequency, or "tone". This twelve-tone system is what we'll be working with in this project.

Twelve Tones

Figure A.64 shows a set of twelve tones that starts with a base frequency of 440 Hz[67]. Each of the tones is obtained by multiplying the preceding term by a constant factor, which we'll call by the Greek letter ρ. If we want to get to twice the original frequency after twelve steps, that means that ρ has to be equal to $\sqrt[12]{2}$ (the twelfth root of two)[68]. If we number the tones starting with zero, then tone number `itone` will have a frequency of $base \times \rho^{itone}$, where `base` is the frequency of tone number zero.

[67] This is the frequency of the A above middle C on a piano.

[68] This is called the 12-tone even-tempered system of tuning.

Notice in Figure A.64 that some of the tones are approximately nice fractions times the base frequency. Tone number 7, for example, is about $\frac{3}{2}$ times the base, and tone 5 is about $\frac{4}{3}$ times the base. Just like pairs of frequencies where one is twice the other, our ears also like to hear sounds that are in small-integer ratios like $\frac{3}{2}$ and $\frac{4}{3}$. This system for picking twelve tones is appealing to musicians because it produces many tones that are approximately small-integer multiples of the base frequency.

The twelve tones in Figure A.64 cover just one octave (plus the first note of the next octave), but we can keep multiplying by ρ to get higher frequencies, or we could start dividing the base frequency by ρ to get lower frequencies, as shown in Figure A.65. Each time we get to a factor of two we've covered another octave, and we're back to a tone that sounds similar to the base frequency. We'll say that the octave that begins with the base frequency is `octave` number 0. In this project, we'll say that frequencies that differ by exactly an octave (or multiple octaves) have the same tone number (`itone`), but different `octave` numbers.

itone	Frequency	Ratio (ρ^{itone})	Fraction
0	440.00	1.00	∼ 1/1
1	466.16	1.06	
2	493.88	1.12	∼ 9/8
3	523.25	1.19	
4	554.37	1.26	∼ 5/4
5	587.33	1.33	∼ 4/3
6	622.25	1.41	
7	659.26	1.50	∼ 3/2
8	698.46	1.59	
9	739.99	1.68	∼ 5/3
10	783.99	1.78	
11	830.61	1.89	∼ 15/8
0	880.00	2.00	∼ 2/1

Figure A.64: Twelve tones, starting with a base frequency of 440 Hz. The table also shows a thirteenth tone, which is just twice the base frequency and is the beginning of the next octave. We'll say that this is tone number 0 again, but in a higher octave.

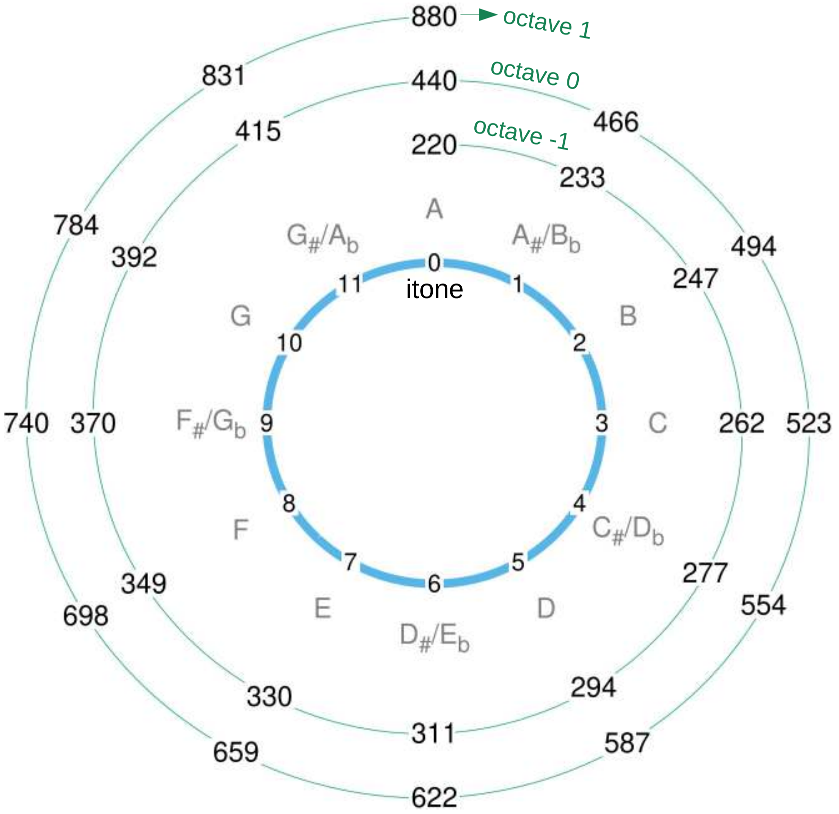


Figure A.65: A spiral of frequencies, starting with a base frequency of 440 Hz and going up and down from there. The frequencies double each time we go around the spiral.

If we choose 440 Hz as our base frequency, the tones in `octave` 0 correspond to the piano keys shown in Figure A.65, where `itone` 0 in `octave` 0 is A above middle C. Figure A.66 shows how these would look on a piano keyboard. The gray letters in Figure A.65 show the standard names for all the tones on a piano. We could choose any base frequency we want, and that would generate a different set of twelve tones with the same ratios between them.

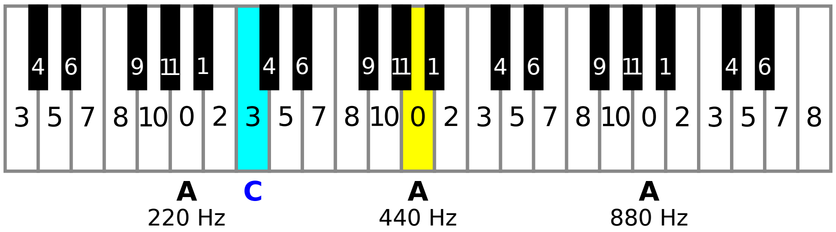


Figure A.66: The center of a piano keyboard, showing middle C and several A keys in different octaves. If we use 440 Hz as the base frequency for our tones, then the A above middle C is `itone` 0 in `octave` 0. The numbers on the keys show the value of `itone` for that key.
Based on a figure from Wikimedia Commons.

Scales and Intervals

Composers don't necessarily use all twelve tones when writing music. They choose a set of tones to make a *scale*. If they do use all twelve of the tones, we call that a *chromatic* scale. We can start with any tone we want. If we started with `itone` 0 on Figure A.65 and went clockwise around the circle picking every tone until we got back to 0, we'd have a "chromatic scale in A". The ratio, or *interval*, between each pair of neighboring notes would be ρ, which we'll call one *step*. Our starting point (`itone` 0 in this case) is called the *tonic* of the scale.

But often composers don't use all of the tones. Look at Figure A.68. In this case, there's an interval of 2 steps between some notes, and an interval of 1 step between others. If we start with `itone` 2 (a B on a piano keyboard) and go from one circled tone to another until we get all the way back to 2, we'll have a "B-major scale" with 7 notes. A *major* scale is any one that has intervals between notes in this pattern.

The type of a scale (chromatic, major, pentatonic, harmonic minor, ... there are a lot!) is determined by the list of intervals between its notes. Figure A.67 shows intervals for several common scales.

Let's use these musical principles to write some programs.

Intervals for a Few Scales

chromatic:	1 1 1 1 1 1 1 1 1 1 1 1
major:	2 2 1 2 2 2 1
pentatonic:	2 2 3 2 3
harmonic minor:	2 1 2 2 1 3 1
major blues:	2 1 1 3 2 3
minor blues:	3 2 1 1 3 2
jazz minor:	2 1 2 2 2 2 1
minor pentatonic:	3 2 2 3 2
ambassel:	1 4 2 1 4
anchihoye:	1 4 1 4 2
bati:	4 2 1 4 1
tezita:	2 1 4 1 4
wenz:	4 1 4 2 1

Figure A.67: List of intervals for several different scales. Notice that for each of these the intervals add up to 12, since that's the total number of steps all the way around the circle of tones. The last five scales are Ethiopian 5-note scales called *kignits*.

Program 1: Simulating a Composer

The first program you'll write is called `simulate.cpp`. It will generate 12 tones starting with a base frequency of 440 Hz. Then it will pick a set of those tones to make a scale, based on a starting tone and a given set of intervals. It will then do a random walk up and down this scale

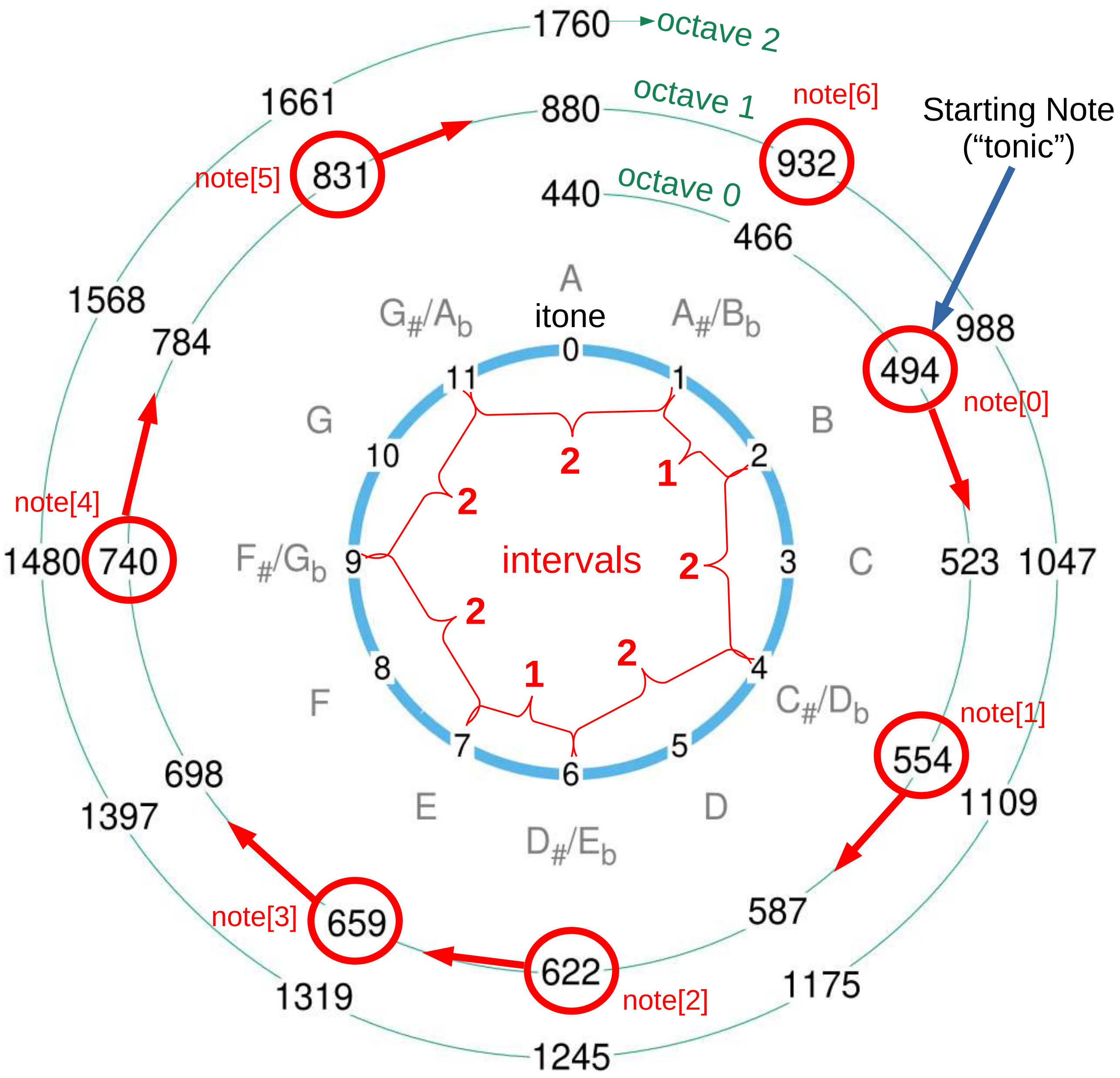


Figure A.68: A B-major scale. It starts with `itone` 2, and steps around the circle in intervals of 2, 2, 1, 2, 2, 2, and 1 steps.

to generate a melody, with some notes skipped at random. Finally, it will write the notes into an output file that can be converted into a playable music file.

Dave Brubeck at the piano.
Source: Wikimedia Commons.

Program A.25 shows part of what this program should look like. Your job is to fill in the missing part, where it says "Insert the rest of the program here!". The program will make use of three functions in the header file named `music.h`, which you can find in the appendix to this project.

The program should be written so that you can run it with command-line arguments like this:

```
./simulate 3 280 0.1 60 major-interval.dat simulate-major-60.dat
```

where the list of command line arguments is:

1. `tonic`: The `itone` number that should be used to start the scale.
2. `npm`: The number of notes per minute that should be played.
3. `restprob`: The probability that the melody will skip a note.
4. `tmax`: How long the melody will last, in seconds.
5. `intervalfile`: A file containing a list of intervals between notes of the scale.
6. `outputfile`: The file that the program's output will be written into.

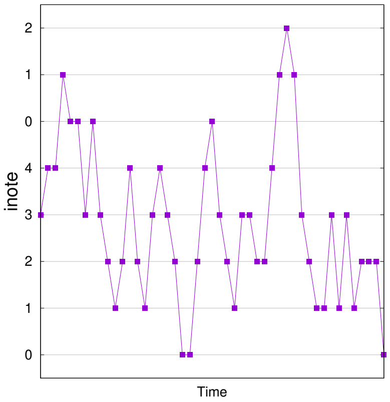


Figure A.69: If we used a 5-note scale, this is what wandering up and down the scale might look like. Here we start at note number 3, then repeatedly move by one or two notes up or down from the note we're currently on.

How to Write the Program

1. Check to make sure the user has supplied enough command-line arguments. If there aren't enough command-line arguments, the program should print a friendly usage message and then stop without trying to do anything else[69].

2. Convert the command-line arguments into the variables `tonic`, `npm`, `restprob`, and `tmax` by using the `atoi` and `atof` functions[70]. The last two command-line arguments (the interval file and output file) don't need to be converted. You can just use them directly, like this[71]:

   ```
   intervalfile = fopen( argv[5], "r" );
   outputfile = fopen( argv[6], "w" );
   ```

3. All of our notes will be of the same length. Calculate the length, in seconds, of a note by dividing 60 seconds by `npm` (notes per minute). Put the result into the variable named `duration`.

4. Next, use a `for` loop to calculate the frequencies of the 12 tones and store those in the array named `tone`. For each value of `itone`, `tone[itone]` should be equal to `base*pow(rho,itone)`, where

[69] See Section 9.16 of Chapter 9 for an example of how to do this.

[70] Notice that `tonic` and `npm` are `int` and `restprob` and `tmax` are `double`.

[71] Notice that we open one file for reading (with `"r"`) and the other for writing (with `"w"`).

Program A.25: simulate.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#include "music.h"

int main ( int argc, char *argv[] ) {

  const int ntones = 12;
  double base = 440;
  double tone[ntones];
  double rho = pow( 2, 1.0/ntones );

  int tonic;
  double note[ntones];
  int nnotes=0;
  int steps;
  int octave = 0;
  double npm;
  double duration;
  double restprob;

  double frequency;
  double minfreq = 200;
  double maxfreq = 1300;

  double t = 0;
  double tmax;

  int position, itone, inote, imove;

  FILE *intervalfile;
  FILE *outputfile;


        // Insert the rest of the program here!

}
```

base and rho are already defined for you in Program A.25.

5. We're going to pick the notes of our scale by starting with the tone we chose as the tonic and then walking clockwise around the spiral path, as shown in Figure A.68. We'll use the variable named position to keep track of where we are on the spiral. At this point in the program, set position = tonic to indicate that this is our starting position..

6. Now use a while loop to read intervals from the interval file you opened earlier[72]. Each line of the file contains one number, the number of tone steps between two notes on our scale[73]. We'll store the frequencies of the tones of our scale in the array named note. The variable ntones tells us how many tones are available[74] and we'll use the variable nnotes (which is initialized to zero) to count how many notes are in our scale.

   The while loop should read numbers from the interval file into the variable named steps. Inside the loop we'll need to do several things:

   (a) First, find the current values of itone and octave by looking at position:

   ```
           itone = position % ntones;
           octave = position / ntones;
   ```

   As we saw in Chapter 4, the % operator gives the remainder after division. Also, since both position and ntones are int, doing position/ntones tells us how many whole multiples of ntones are in position.

   (b) Next, use these to find the frequency of this note:

   ```
   note[nnotes] = tone[ itone ]*pow(2,octave);
   ```

   (c) Finally, add steps to the value of position, and add one to the value of nnotes.

   Keep doing this until you've read all of the numbers from the interval file.

7. Next, we need to pick a random note from our scale to start our melody. The header file music.h contains a function named randminmax that will help us do that. It generates a random number between given minimum and maximum values. We'll use the variable position again. Start by setting position to a random note on our scale:

   ```
   position = randminmax( 0, nnotes-1 );
   ```

[72] See Program 5.4 in Chapter 5 for an example of reading numbers from a file.

[73] See Figure A.68.

[74] This is set to 12 at the top of our program. If we wanted to use a musical system with a different number of tones, we could change this number to something else.

8. Now we're ready to generate our melody. We'll use the variable `t` to keep track of the elapsed time in our melody, and the variable `frequency` to keep track of the frequency of the current note.

   To prevent damaging our ears or generating notes too low-pitched to hear, the program has two variables named `maxfreq` and `minfreq`. If our notes wander above `maxfreq` we'll reduce it by one octave, cutting the frequency in half. If they wander below `minfreq` we'll increase it by one octave, doubling the frequency.

   With all that in mind, create a `while` loop that keeps repeating for as long as `t` is less than `tmax`.

   Here are the things that need to be done inside the loop:

   (a) First, find the values of `inote` and `octave` based on `position`. We did something similar in the previous while loop, but this time it's trickier because we'll sometimes be dealing with negative numbers. Our path can wander up or down from our starting point, so sometimes `position` will be negative (see Figure A.70). The `emod` function is defined in `music.h`. Unlike the mathematical modulo operator, most computer languages use a modulo operator that gives a negative answer for `n%m` when `n` is negative. The `emod` function does a modulo operation that always gives a positive answer[75], which is what we want because `inote` should always be positive.

   To find `octave` we want to divide `position` by `nnotes` and round down. If `position` were always positive, we could just say `position/nnotes`, since both variables are `int`. But for negative values of `position` this would round *up*, making `-1/7` equal to zero, for example. To always round down, we can use the `floor` function, which is one of C's standard math functions.

   So, with all that in mind here's how we'll find the values for `inote` and `octave`:

   ```
   inote = emod(position,nnotes);
   octave = floor( (double)position/nnotes );
   ```

   (b) Next, check to see if we're supposed to skip this note. We do that by generating a random number between zero and one, using the `rand01` function from `music.h`, and checking whether it's less than `restprob`. If it is, we set `frequency` to 0.

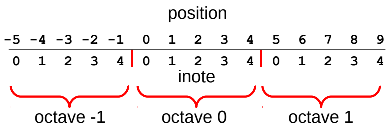


Figure A.70: Here are three octaves of an `nnotes` = 5 scale. Notice that if `position` is 6, then `inote` is 1 and `octave` is 1. This is the same thing we would have gotten by saying `inote = position % nnotes` and `octave = position/nnotes`. But if `position` is -2, `inote` is 3 and `octave` is -1. Because of the negative `position`, the `%` and `/` operators by themselves won't give us the right values for `inote` and `octave` in this case.

Ahmad Jamal at the piano.

Source: Wikimedia Commons.

[75] This is called a Euclidean modulo, which is the modulo operator usually used in mathematics.

Otherwise, we set `frequency` to

```
note[inote] * pow( 2, octave )
```

and check to make sure the frequency isn't too low or too high. If `freqeuency` is above `maxfreq`, reduce `position` by `nnotes` and divide `frequency` by 2. If it's below `minfreq`, increase `position` by `nnotes` and double `frequency`.

Shirley Horn at the piano.
Source: Wikimedia Commons.

(c) Now we're ready to write this note into the output file. We'll write two numbers for each note: The duration of the note, in seconds (given by the variable `duration`) and the frequency of the note (given by `frequency`). They should be written as two numbers separated by a space, and followed by a line break[76].

(d) Next we need to decide how far and in which direction we'll walk to get to the next note. We'll move up or down the scale at random by one or two notes. To decide how many notes to move, we can use the `randminmax` function again, like this:

```
imove = randminmax( -2, 2 );
```

(e) Then add `imove` to `position`.

(f) The last thing in the `while` loop should be a line that adds this note's duration to the total elapsed time:

```
t += duration;
```

Running the Program

After you've finished writing and compiling the program you'll need some interval files to try it out with. If you have a nice instructor, they might give you these files. Otherwise, you can copy them from the appendix to this project.

When you have the interval files, try running your program a few times with different ones, like this:

```
./simulate 0 280 0.1 60 major-interval.dat simulate-major-60.dat
./simulate 0 280 0.1 60 harmonic-minor-interval.dat simulate-harmonic-minor-60.dat
./simulate 0 280 0.1 60 pentatonic-interval.dat simulate-pentatonic-60.dat
./simulate 0 280 0.1 60 chromatic-interval.dat simulate-chromatic-60.dat
```

You can get a sense of what the melodies in the output files are like by plotting them with *gnuplot*. For example:

```
  plot "simulate-major-60.dat" using 2 with lines
```

[76] See Program 5.3 in Chapter 5 for an example of writing numbers into a file.

Figure A.71 shows what you might see.

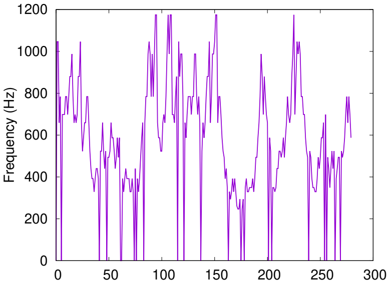


Figure A.71: An output file from the `simulate` program, graphed with *gnuplot*.

But of course what we'd really like to do is *hear* the melody. For that, you can use the `makewav` program in the appendix. You can run it like this:

```
./makewav simulate-major-60.dat major-60.wav
```

This will make a `.wav` file that your computer should be able to play as sound. Try listening to melodies generated with different scales and different starting tones.

Program 2: Analyzing a Melody

Now that you've made some melodies, it would be nice to check your first program's output to make sure it's doing what you expect. One way to do this would be to look at the ratios between successive notes. These should always be multiples of ρ, the ratio between the twelve tones we started with.

Fats Waller at the piano.

Source: Wikimedia Commons.

The second program you'll write is called `analyze.cpp`. It will read the files created by your first program and make a list of the number of steps between each pair of successive notes. Program A.26 shows how the program should start. You'll just need to fill in the missing part.

If frequency f_2 is higher than f_1 by n steps, then the following should be true:

$$\frac{f_2}{f_1} = \rho^n \tag{A.3}$$

To find n we can take the log of both sides:

$$log(\frac{f_2}{f_1}) = n \times log(\rho) \tag{A.4}$$

and we can rearrange this to find n:

$$n = \frac{log(\frac{f_2}{f_1})}{log(\rho)} \tag{A.5}$$

You'll find that if f_2 is *smaller* than f_1 by n steps, it will just change the sign of n. So, if we look at the absolute value of n it will tell us how many steps up or down we need to go to get from one note to the other.

If our `simulate` program is working correctly, all the values of n should be whole numbers.

Program A.26: analyze.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
int main ( int argc, char *argv[] ) {
  const int ntones = 12;
  double duration, frequency;
  double oldfreq = 0;
  double ratio;
  double rho = pow( 2, 1.0/ntones );
  double steps;
  FILE *input;
  FILE *output;


        // Insert the rest of the program here!

}
```

Thelonious Monk at the piano.

Source: Wikimedia Commons.

The program should be written so that you can run it with command-line arguments like this:

```
./analyze input output
```

where `input` is the name of one of the files produced by your `simulate` program, and `output` is the name of the file that this new program will create.

How to Write the Program

1. Check to make sure the user has supplied enough command-line arguments. If there aren't enough command-line arguments, the program should print a friendly usage message and then stop without trying to do anything else[77].

2. For this program, the two command-line arguments don't need to be converted in any way. You can just use them directly, like this:

   ```
   input = fopen( argv[1], "r" );
   output = fopen( argv[2], "w" );
   ```

3. Now use a `while` loop to read lines from the input file you opened earlier[78]. Each line of the file contains two numbers with decimal places: the duration of a note, and the frequency of the note. Put those numbers into the variables named `duration` and `frequency`.

4. Inside the `while` loop you'll need to do several things:

   (a) Some of the notes in the data file will have a frequency of zero. These are the notes we skipped. We want to also skip them in this second program, so you'll need to check to see if `frequency ==`

[77] See Section 9.16 of Chapter 9 for an example of how to do this.

[78] See Program 5.4 in Chapter 5 for an example of reading numbers from a file.

0. If it is, then use a `continue` statement to go back to the top of the loop and get the next line from the file.

(b) Our goal is to find ratios of pairs of notes, so we need to have two notes to compare. To do that, we'll store the frequency of the note we previously read in the variable named `oldfreq`. Then we can look at the ratio of `frequency` to `oldfreq`. You'll notice that `oldfreq` has an initial value of zero, so we'll use this as an indicator that we haven't yet read two frequencies. We're also uninterested in cases where the frequency hasn't changed. With all of that in mind, we'll now need an `if` statement that starts like this:

```
if ( frequency != oldfreq && oldfreq != 0 ) {
```

and inside this `if` statement we'll need to do the following:

i. Find the ratio of `frequency` to `oldfreq` and put it into the variable named `ratio`.

ii. Calculate the absolute value of n (the number of steps) from Equation A.5 above, like this:

```
steps = fabs( log(ratio)/log(rho) );
```

iii. Print the number of steps into the output file, followed by a line break.

(c) At the bottom of the `while` loop, save the current frequency in the variable `oldfreq` by saying:

```
oldfreq = frequency;
```

Count Basie at the piano.
Source: Wikimedia Commons.

After you've written and compiled the program, try running it with some of the output files from your first program. For example:

```
./analyze simulate-chromatic-60.dat analyze-chromatic-60.dat
./analyze simulate-major-60.dat analyze-major-60.dat
./analyze simulate-pentatonic-60.dat analyze-pentatonic-60.dat
```

If you graph the output files with *gnuplot* you should see graphs like Figure A.72. As you can see, there are only whole numbers of steps between the notes.

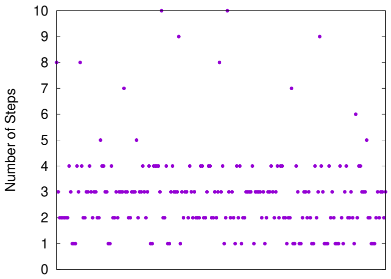


Figure A.72: Output from the `analyze` program, plotted with *gnuplot*.

Program 3: Counting Notes

The output from the `analyze` program tells us that there are integer numbers of steps between notes, but it doesn't tell us how often we take 1 step, 2 steps, 3 steps, and so forth. To find that out, we can make a histogram. That's what your third program will do.

This third program will be called `visualize.cpp` and it will read output files created by the `analyze` program and make histograms from the data in them. Program A.27 shows how the program should start. You just need to fill in the missing part.

Bill Evans at the piano.

Source: Wikimedia Commons.

Program A.27: visualize.cpp

```
#include <stdio.h>
#include <stdlib.h>
int main ( int argc, char *argv[] ) {
  const int nbins=100;
  int bin[nbins] = {0};
  double min, max;
  double binsize;
  double steps;
  int binno;
  int overunderflow=0;
  int i;
  FILE *input;
  FILE *output;

        // Insert the rest of the program here!

}
```

How to Write the Program

Like the preceding programs, this one will expect parameters on its command line, and should complain and exit if it doesn't get the proper number of parameters. Its usage will be:

```
./visualize min max input output
```

where `min` and `max` are the minimum and maximum number of steps that can be recorded in the histogram; `input` is the name of a file that was produced by `analyze.cpp`; and `output` is the name of a file into which the new program will write the histogram data. When plotted, the data should look like those shown Figure A.73 on page 613.

To make the histogram, the program should proceed as follows:

1. Check the number of command-line arguments, and use `atof` to set the values of `min` and `max`. The input and output files can be opened like this:

```
input  = fopen(argv[3],"r");
output = fopen(argv[4],"w");
```

2. Then, determine the binsize, like this:

```
binsize = (max-min)/nbins;
```

3. Next, use a while loop to read data from the input file[79]. Your while loop should read one number from the input file each time it goes around. It should stick the number into the variable named steps.

Abdullah Ibrahim at the piano.

Source: Wikimedia Commons.

4. Inside the loop, determine which bin each steps value belongs in, and increment that bin. Be sure to keep a count of the number of over/underflows, as Program 7.1 does[80]. Since the range of our histogram is min to max, the bin number will be:

```
binno = (steps-min)/binsize;
```

5. After processing all of the input data, write the histogram data into the output file. For each bin of the histogram, write two numbers separated by a single space: the steps value represented by that bin[81], and the value of bin[i]. The steps value can be calculated from the bin number, like this:

```
steps = min + binsize*(0.5+i);
```

where i is the bin number.

6. Finally, at the bottom of the output file, write a line beginning with a # that tells how many overflows or underflows were seen.

Compile the program and run it like this, using some data files created by your analyze program:

```
./visualize 0 13 analyze-chromatic-60.dat visualize-chromatic-60.dat
./visualize 0 13 analyze-major-60.dat visualize-major-60.dat
./visualize 0 13 analyze-pentatonic-60.dat visualize-pentatonic-60.dat
```

The output files can be graphed with *gnuplot* commands like this:

```
plot "visualize-chromatic-60.dat" with impulses
```

This should produce graphs like those shown in Figure A.73.

It's possible to identify the scale a melody uses by looking at these histograms. For example, the pentatonic scale has no intervals of 1 step[82], so there are no step values of 1 in the pentatonic histogram.

[79] See Chapter 5 for information about reading data from files.

[80] See Section 7.1 of Chapter 7 for information about making histograms.

[81] Note that this is different from Program 7.1, where we just printed the bin number as the first column in the output file.

[82] see Figure A.67 on page 601

On the other hand, all of the intervals of the chromatic scale are 1 step, so if our melody goes up or down the scale by 1 or 2 notes, we'd expect to only see intervals of 1 or 2 steps in the chromatic histogram. This is mostly the case, but there are also a few values in the histogram's 3 column. This is because our simulation program sometimes intentionally skipped a note, which could leave a larger than expected gap between the notes we hear. Also, if you look closely you might see some data in the 10 or 11 column. These happened when the simulation program was at note 0 or 1 and stepped down from there, going to a high note in a lower octave.

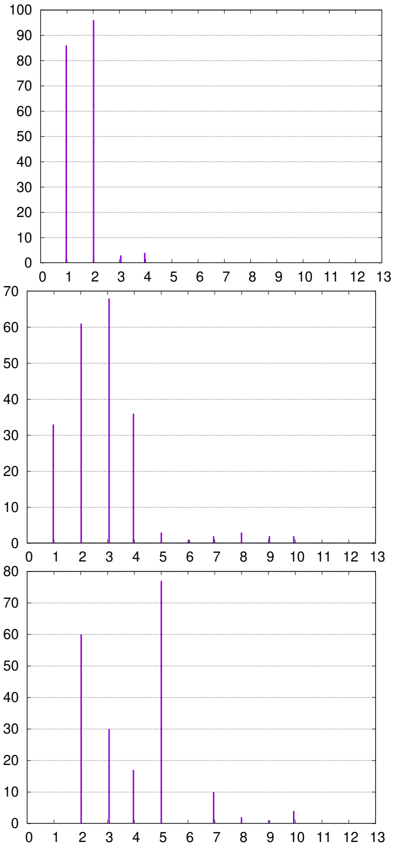


Figure A.73: Histograms made with the `visualize` program. The top graph shows the result for a melody produced using a chromatic scale. The middle graph is a major scale. The bottom graph is a pentatonic scale.

Conclusion

Hopefully you've created some melodies to inspire your songwriter friend.

There are many enhancements you could add to your `simulate` program. For example, you could take a short series of notes and repeat it, maybe with modifications like raising or lowering the last note every other time it's repeated, or raising or lowering all the notes by an octave. Also, instead of only going up or down by one or two notes, you could sometimes, at random, leap to a really high or low note.

You might think about how `simulate.cpp` could be modified so to create melodies that have more than one note playing at the same time. You'd need to think about which notes sound good together, and maybe do some more research into the mathematics of chords. You'd also need to modify `makewav.cpp`, if you wanted to hear the melodies.

For more information about the mathematics of music, I highly recommend the series of videos that Norman Wildberger has been doing recently on his Insights into Mathematics Youtube channel.

Hank Jones at the piano.

Source: Wikimedia Commons.

Appendix

- **The `music.h` header file**
  Here's the `music.h` header file that you'll need for writing `simulate.cpp`:

Program A.28: music.h

```
// Random number between 0 and 1:
double rand01 () {
  static int needsrand = 1;
  if ( needsrand ) {
    srand(time(NULL));
    needsrand = 0;
  }
  return ( rand()/(1.0+RAND_MAX) );
}

// Random number between min and max:
double randminmax( int min, int max ) {
  return ( min + (int)( (max-min+1)*rand01() ) );
}

// Euclidean modulus (always positive):
int emod ( int n, int modulus ) {
  return (   (n%modulus + modulus)%modulus );
}
```

- **Some Interval Files**
  Here are the contents of three files that give the interval sequence for a major scale, a harmonic minor scale, and a pentatonic scale. As you can see, they're just lists of the numbers of steps between notes on each scale. You can create other files using the intervals listed in Figure A.67.

major-interval.dat

```
2
2
1
2
2
2
1
```

harmonic-minor-interval.dat

```
2
1
2
2
1
3
1
```

pentatonic-interval.dat

```
2
2
3
2
3
```

You can also create a new scale by taking the top interval from the major scale and moving it to the bottom. This is called the *Dorian* mode. By continuing to move the top interval to the bottom, you can produce other scales. The next one is *Phrygian*, then *Lydian*, *Mixolydian*, *Aeolian*, and *Locrian*. If you do it one more time, you're back to where you started, which is called the *Ionian* mode.

- **The `makewav` program**
  Here's the `makewav` program that you can use to convert the output from your `simulate` program into playable `.wav` files. In order to compile it, you'll also need to fetch two other files:

  ```
  https://www3.nd.edu/~dthain/courses/cse20211/fall2013/wavfile/wavfile.c
  https://www3.nd.edu/~dthain/courses/cse20211/fall2013/wavfile/wavfile.h
  ```

  You can use *wget* or *curl* to download these. After you have them and the `makewav.cpp` file below, you can compile the `makewav` program by typing:

  ```
  g++ -Wall -O -o makewav makewav.cpp wavfile.c
  ```

Program A.29: makewav.cpp

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#include "wavfile.h"

double safefreq ( double frequency ) {
  double minfreq = 200;
  double maxfreq = 1300;
  if ( frequency > maxfreq || frequency < minfreq ) {
    return(0);
  } else {
    return(frequency);
  }
}

int main( int argc, char *argv[] ) {
  double duration, frequency;

  double t = 0;
  double tsample = 1.0/WAVFILE_SAMPLES_PER_SECOND;
  int nsamples;

  short amplitude;
  int volume = 8000;

  int i;

  double oldphase = 0, phase;

  FILE *input;
  FILE *output;


  input = fopen( argv[1], "r" );
  if(!input) {
```

```
    printf("couldn't open %s for writing: %s",argv[1],strerror(errno));
    exit(1);
  }


  output = wavfile_open( argv[2] );
  if(!output) {
    printf("couldn't open %s for writing: %s",argv[2],strerror(errno));
    exit(1);
  }

  while ( fscanf( input, "%lf %lf", &duration, &frequency ) != EOF ) {
    frequency = safefreq( frequency );

    nsamples = duration/tsample;
    phase = oldphase - frequency*t*2*M_PI;
    for ( i=0; i<nsamples; i++ ) {
      amplitude = volume*sin(frequency*t*2*M_PI + phase);
      oldphase = frequency*t*2*M_PI + phase;
      wavfile_write(output,&amplitude,1);
      t += tsample;
    }
  }

  wavfile_close(output);
  fclose(input);
}
```

B. Installing Necessary Software

B.1. For Microsoft Windows

Windows 10:

Microsoft has released a set of tools for Windows 10 that allows you to install a complete development environment, including the tools used in this book, fairly easily. The first thing you'll need to install is called "Windows Subsystem for Linux" (WSL). You'll find up-to-date instructions for installing it here:

https://docs.microsoft.com/en-us/windows/wsl/install

At some point during the installation you might be asked which Linux distribution to use. Pick the latest available version of "Ubuntu".

After you install WSL, there are a few more things you'll need to do:

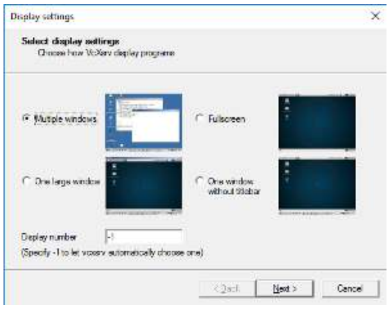
1. Update and install software in the Ubuntu App:
   Start the app and type the following commands in the app's window:
   ```
   sudo apt update
   sudo apt -y upgrade
   sudo apt -y install g++ nano gnuplot-x11
   sudo apt -y reinstall gnome-icon-theme
   ```
   The first command will ask for the user name and password you entered while setting up WSL. If any of the commands asks you about restarting services, answer "yes".
   This will install the specific software (*g++*, *nano*, and *gnuplot*) used in this book.
2. Next, type the following commands:

```
wget http://tinyurl.com/practical-c/wsl-fix.sh
sh wsl-fix.sh
source .bashrc
```

3. In order to use this version of *gnuplot* you'll need to install one more piece of Windows software, called an "X server". This allows the tools in the development environment (installed in the steps above) to display graphics on your screen. To install it, download and install VcXsrv from here:
   https://sourceforge.net/projects/vcxsrv/
4. After you've installed VcXsrv, click the Start button, and type "xlaunch" and press the Enter key. A window like Figure B.1 should appear. Keep clicking "Next" until you get to the dialog box shown in Figure B.2. Put a check mark in all of the boxes in this section.
   Then continue clicking "Next" until you get to the dialog box shown in Figure B.3.
   Click "Save Configuration" and save the configuration as "config.xlaunch" on your desktop.
   Now hold down the "Windows" key and type R, press "Enter", then type:

   ```
   shell:startup
   ```

   and press the Enter key. This will open up your startup folder. Drag the "config.xlaunch" icon from your desktop into this folder.
5. Now restart your computer. You should be able to get a command window by clicking "Ubuntu" in the Start Menu. All the work in this book can be done in this window.



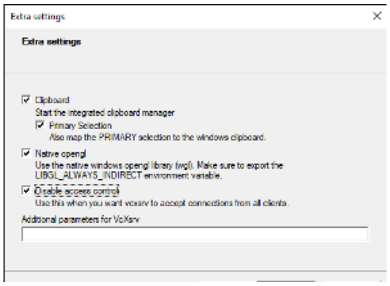
Display settings

Select display settings
Choose how VcXsrv display programs

● Multiple windows
○ Fullscreen
○ One large window
○ One window without titlebar

Display number: -1
(Specify -1 to let vcxsrv automatically choose one)

Figure B.1: Running xlaunch.

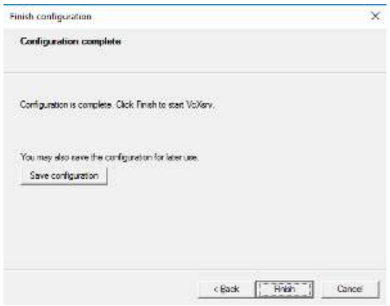


Extra settings

Extra settings

☑ Clipboard
Start the integrated clipboard manager
☑ Primary Selection
Also map the PRIMARY selection to the windows clipboard.
☑ Native opengl
Use the native windows opengl library (wgl). Make sure to export the LIBGL_ALWAYS_INDIRECT environment variable.
☑ Disable access control
Use this when you want vcxsrv to accept connections from all clients.
Additional parameters for VcXsrv

Figure B.2: "Extra Settings" screen.



Finish configuration

Configuration complete

Configuration is complete. Click Finish to start VcXsrv.

You may also save the configuration for later use.

Figure B.3: Saving xlaunch configuration.

B.2. For Linux

On Debian, Ubuntu, Mint and similar distributions, type:

```
sudo apt update
sudo apt -y upgrade
sudo apt -y install g++ nano gnuplot
```

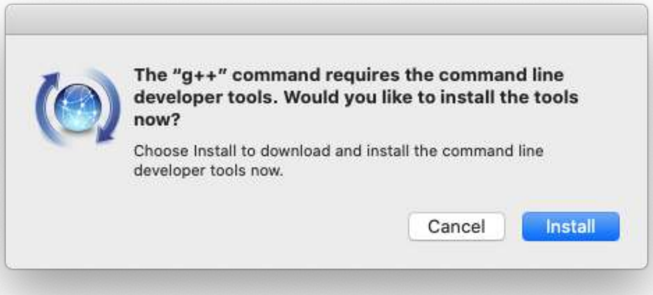
On Fedora, CentOS, Red Hat and similar distributions, type:

```
sudo yum install gcc-c++ nano gnuplot
```

B.3. For Apple MacOS

For Mac users, Apple includes many of the tools you'll need, but they might need to be "activated". You'll also need *gnuplot* and *Xquartz*, upon which *gnuplot* depends.

1. To get a command window, click any blank spot on your desktop background, then go to the "Go" menu at the top of the screen and select Utilities->Terminal.
2. Inside the terminal window, type *g++*. The first time you do this you'll see a message like the one below.



The "g++" command requires the command line developer tools. Would you like to install the tools now?

Choose Install to download and install the command line developer tools now.

Click "Install" to install the command line developer tools. Now you should be able to use the *g++* command as we do in this book.
3. In order to use *gnuplot* under OS X, you'll also need to install two more things. The first is *XQuartz*, which you can get here: http://xquartz.macosforge.org/landing/
4. **IMPORTANT:** After you've installed *XQuartz*, you must log out of your computer and log back in to complete the installation. (If you don't do this, gnuplot may not install or work correctly.)
5. The last thing to install is *gnuplot* itself, which you can get here: https://csml-wiki.northwestern.edu/index.php/Binary_versions_of_Gnuplot_for_OS_X

Download the current version of *gnuplot* from the site above.

6. **IMPORTANT:** After you've downloaded the file, hold down the Ctrl key while clicking on it. If you don't hold down the Ctrl key, the computer might refuse to run the installer. Then proceed to install the package as usual.

You should now be able to use the *g++*, *nano*, and *gnuplot* in your terminal window.

C. Getting Example Data Sets

C.1. Star Data (HYG Database)

David Nash, amateur astronomer, has assembled a database of nearby stars that combines data from three sources:

- The Hipparcos satellite's massive survey of millions of stars
- The Yale Bright Star Catalog, containing data for about 10,000 stars
- The Gliese Catalog of Nearby Stars, containing about 4,000 stars.

Nash combined the nearby stars in these databases to form the HYG database (for "Hipparcos, Yale, Gliese").

Figure C.1: The Hipparcos satellite before launch.

Source: Wikimedia Commons

For one of the exercises in Chapter 5 you'll need to download the HYG database and create a new, smaller, database from it. The resulting file will be called `stars.dat` and it's used in Exercise 31. Here's how to get the database and create `stars.dat` from it. The process will involve a couple of mysterious commands that I won't explain, but feel free to do some research on your own to find out what they do. The steps to create `stars.dat` are:

1. Fetch the HYG database. There are two tools that let you do this easily. Use whichever tool is installed on the computer you're using. The first tool is `wget`. The `wget` command lets you download files from a web site without needing to use a web browser. Here's how to use `wget` to download the HYG database;

   ```
   wget https://raw.github.com/astronexus/HYG-Database/master/hygdata_v3.csv
   ```

   If the computer you're using doesn't have `wget`, it probably has a similar tool named `curl`. Here's the `curl` command for downloading the database:

   ```
   curl -L -O https://raw.github.com/astronexus/HYG-Database/master/hygdata_v3.csv
   ```

2. Extract the part of the data that we'll be using in Exercise 31. Note that this is one big, long command without any line breaks. Every character in it is important, so type carefully. (If you can cut-and-

paste the command, it's a good idea to do so.)

```
cat hygdata_v3.csv | grep -v -E 'Sol|^id' | awk -F, '{print $18,$19,$20}' > stars.dat
```

What does this command do? First, it uses the *grep* command to exclude two rows of data: a row of column headers, and the row for our Sun (which is included in the data just like other local stars). Second, it uses the *awk* command to select only three columns: just the columns that hold the x, y, and z coordinates of each star.

That's it! You now have the `stars.dat` database, and you're ready for Exercise 31.

You might want to play around with other data in the HYG database. If so, you can find a description of the data it contains here:

https://github.com/astronexus/HYG-Database

C.2. Normally-Distributed Data

Chapter 7 uses the file `energy.dat` for several exercises. This file contains simulated energy measurements from a scintillation counter. The energy values are "normally" distributed, meaning that when we make a histogram of the values it has the shape of a Normal distribution.

Figure C.2: Carl Friedrich Gauss, who studied the Normal distribution extensively.

Source: Wikimedia Commons

You can generate `energy.dat` by compiling Program C.1 and running it like this:

```
./mkenergy > energy.dat
```

Program C.1 uses a technique called the Box-Muller Transform to generate normally-distributed numbers. It defines a function named `normal` that takes two arguments (the mean of the distribution and its standard deviation) and returns a single pseudo-random number. (You'll understand how to create C functions after reading Chapter 9.) The program's `main` function just uses `normal` to generate 100,000 numbers. By changing the mean and standard deviation, you can change the distribution of the numbers. Try it, it's fun!

Program C.1: mkenergy.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
double normal(double mean, double sigma) {
  // Use Box-Mueller Tranform to generate
  // normally-distributed numbers.
  const double epsilon = 1e-9;
  const double two_pi = 2.0*M_PI;
  static double z0, z1;
  static int generate=1;
  static int initialized=0;
  double u1, u2;

  if (!initialized) {
    srand(time(NULL));
    initialized = 1;
  }

  if (!generate) {
    generate = 1;
    return z1 * sigma + mean;
  } else {
    do
      {
        u1 = rand() * (1.0 / RAND_MAX);
        u2 = rand() * (1.0 / RAND_MAX);
      }
    while ( u1 <= epsilon );

    z0 = sqrt(-2.0 * log(u1)) * cos(two_pi * u2);
    z1 = sqrt(-2.0 * log(u1)) * sin(two_pi * u2);
    generate = 0;
    return z0 * sigma + mean;
  }
}
int main () {
  int i;
  for ( i=0; i<100000; i++ ) {
    printf ("%lf\n",normal(35.0,2.5) );
  }
}
```

Figure C.3: Statistician George E.P. Box, one of the inventors of the Box-Muller transform.

Source: Wikimedia Commons

C.3. Census Data (American Community Survey)

In addition to the decennial census mandated by the U.S. constitution, the Census Bureau conducts many other surveys. One of these is the ongoing *American Community Survey* (ACS), which gathers data about how Americans live in their communities. Data from the ACS help local governments decide how to spend their money.

Figure C.4: A U.S. census worker transcribing data onto punched cards during the 1950s.
Source: *Wikimedia Commons*

ACS data can be downloaded from the Census Bureau's web site. In order to protect the identities of the citizens who respond to the survey, only an anonymized sample of the data (called a "Public Use Microdata Sample" or "PUMS") is provided. These data sets are available here:

https://www.census.gov/programs-surveys/acs/data/pums.html

Exercise 40 on page 252 uses a data file named `census.dat` derived from the ACS data collected during the years 2011 through 2013. Here's how to make it:

1. First, as in Section C.1 above, you'll need to fetch some data from a web site. If your computer has the `wget` command, you can do it this way:

   ```
   wget http://www2.census.gov/acs2013_3yr/pums/csv_hus.zip
   ```

   otherwise, you can use the `curl` command like this:

   ```
   curl -L -O http://www2.census.gov/acs2013_3yr/pums/csv_hus.zip
   ```

2. The file you downloaded is named `csv_hus.zip`. This file has several data sets packed inside it, so the next step is to unpack them. You can do this by using the following command:

   ```
   unzip csv_hus.zip
   ```

   This will extract five files:

   ```
   ss13husa.csv
   ss13husb.csv
   ss13husc.csv
   ss13husd.csv
   ACS2011-2013_PUMS_README.pdf
   ```

   The last file contains documentation describing the data, and the other files contain the actual data, broken into four parts.
   If you looked inside one of the `csv` files you unpacked, you'd see that each line of the files was just a list of values separated by commas. The "`csv`" in the file name stands for "comma-separated values". The data is organized in rows and columns. Each row represents

a single household, and each column is a particular kind of data about that household (number of children or household income, for example). The top row is a comma-separated list of abbreviations telling us what each column represents.

3. To produce our census.dat file we're going to extract just a few of these columns. We could do this using awk and grep, as we did in Section C.1, but this is a book about C programming, so let's use a C program to do it this time.

   The program datafilter.cpp (Program C.2) contains a lot of stuff that you haven't seen before unless you've already finished reading this book. Much of it will become clear after you reach Chapter 8, and most of the rest after you read Chapter 9. The only parts that we won't cover in this book are the malloc and free functions. You'll have to learn about those in a different book, or do some research on your own.

   For now, just save this program as datafilter.cpp and compile it by typing "g++ -Wall -o datafilter datafilter.cpp".

4. The datafilter program needs a configuration file to tell it what to do. Using *nano*, create a file called census.conf containing the following lines:

   ```
   ,
   -1
   NRC
   ACR
   BDSP
   FINCP
   FULP
   GASP
   GRNTP
   ```

   Notice that the first line is just a comma on a line by itself. This tells datafilter that the columns in our data file will be separated by commas. The second line of the file tells datafilter that it should replace any missing data values with "-1". The rest of the lines are a list of columns that datafilter should select. These are the names that appear in the top row of each csv file.

Program C.2: datafilter.cpp

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
int main( int argc, char *argv[] )
{
  const int maxcolumns = 100;
  const int maxlength = 4096;
  char *output[maxcolumns];
  char *wanted[maxcolumns];
  int wantedfield[maxcolumns];
  int nwanted;
  char line[maxlength];
  char *position;
  char delimiter[maxlength];
  char blankvalue[maxlength];
  char *word;
  int i, field;
  FILE *input;
  FILE *setup;

  // Check syntax:
  if ( argc > 3 ) {
    fprintf ( stderr, "Syntax: %s datafile configfile\n", argv[0] );
    exit (1);
  }

  // Open data file:
  if ( !strcmp( argv[1], "-" ) ) {
    fprintf ( stderr, "Reading data from stdin.\n" );
    input = stdin;
  } else {
    if ( ( input = fopen ( argv[1], "r" ) ) ) {
      fprintf ( stderr, "Reading data from %s.\n", argv[1] );
    } else {
      fprintf ( stderr, "Error opening \"%s\": %s\n", argv[1], strerror(errno) );
      exit(1);
    }
  }

  // Open configuration file:
  if ( ( setup = fopen ( argv[2], "r" ) ) ) {
    fprintf ( stderr, "Reading setup from %s.\n", argv[2] );
  } else {
    fprintf ( stderr, "Error opening \"%s\": %s\n", argv[2], strerror(errno) );
    exit(1);
  }


  // Read delimiter:
  fgets( delimiter, 10, setup );
  delimiter[strcspn(delimiter, "\r\n")] = 0;

  // Read blank value:
  fgets( blankvalue, maxlength, setup );
  blankvalue[strcspn(blankvalue, "\r\n")] = 0;


  // Read fields:
  nwanted = 0;
  for ( i=0; i<maxcolumns; i++ ) {
    if ( !fgets ( line, maxlength, setup ) ) { // Break at EOF.
      break;
    }
    line[strcspn(line, "\r\n")] = 0;
    wanted[i] = (char *)malloc( strlen( line ) + 1 );
    snprintf( wanted[i], strlen( line ) + 1, "%s", line );
    nwanted++;
  }

  // Close configuration file:
  fclose ( setup );

  // Read header:
  fgets( line, maxlength, input);
  line[strcspn(line, "\r\n")] = 0;
  position = line;
  field = 0;
  while (position != NULL) {
    word = strsep(&position, delimiter);
    for ( i=0; i<nwanted; i++ ) {
      if ( !strcmp( word, wanted[i] ) ) {
        wantedfield[i] = field;
      }
    }
    field++;
  }

  // Read data:
  while ( fgets( line, maxlength, input) ) {
    line[strcspn(line, "\r\n")] = 0;
    position = line;
    field = 0;
    while (position != NULL) {
      word = strsep(&position, delimiter);
      for ( i=0; i<nwanted; i++ ) {
        if ( field == wantedfield[i] ) {
          output[i] = (char *)malloc( strlen(word) + 1 );
          if ( strlen(word) ) {
            snprintf( output[i], strlen(word) + 1, word );
          } else {
            snprintf( output[i], strlen(blankvalue) + 1, blankvalue );
          }
        }
      }
      field++;
    }
    for ( i=0; i<nwanted; i++ ) {
      printf ( "%s ", output[i] );
      free ( output[i] );
    }
    printf ("\n");
  }

  if ( input != stdin ) {
    fclose ( input );
  }
}
```

5. Now we're ready to create `census.dat`. Type the following commands to do it:

```
./datafilter ss13husa.csv census.conf > census.dat
./datafilter ss13husb.csv census.conf >> census.dat
./datafilter ss13husc.csv census.conf >> census.dat
./datafilter ss13husd.csv census.conf >> census.dat
```

Each of these lines processes one of the `csv` data files and appends the columns extracted from it onto the end of the file `census.dat`. Seven columns are extracted from the original data. These columns are[1]:

0	NRC	Number of related children in household
1	ACR	Lot size, in acres
2	BDSP	Number of bedrooms
3	FINCP	Family income
4	FULP	Annual fuel cost
5	GASP	Monthly gas cost
6	GRNTP	Monthly rent

If you'd like to do further research with this data you can find a complete description of each of the columns in the `csv` files here:

http://www2.census.gov/programs-surveys/acs/tech_docs/pums/data_dict/PUMS_Data_Dictionary_2011-2013.txt

[1] Notice that we've numbered them like the elements of a C array, starting with zero instead of one.

D. Some Notes About **gnuplot**

D.1. What is *gnuplot*?

gnuplot is a general-purpose plotting/graphing program that is quite flexible and surprisingly powerful. It can graph 2- and 3-d functions defined by the user. *gnuplot* also has a wide array of built-in functions, covering trigonometry, as you might expect, but also extending to bessel functions, the gamma function, the error function (erf) and many others.

gnuplot can also plot 2- and 3-d data. Data can be read in either ascii or binary format. Since *gnuplot* allows you to specify the layout of the data file, it can accommodate many different file formats.

gnuplot is cross-platform (Linux, Windows and OS X), and it's free and open-source.

gnuplot is command-line driven. This means that you can write scripts and re-use them later, and it makes it possible to easily tell other people what you've done. The program also has very good built-in help. Just type ˝help˝ at the *gnuplot* command prompt, and you can browse through documentation for every feature.

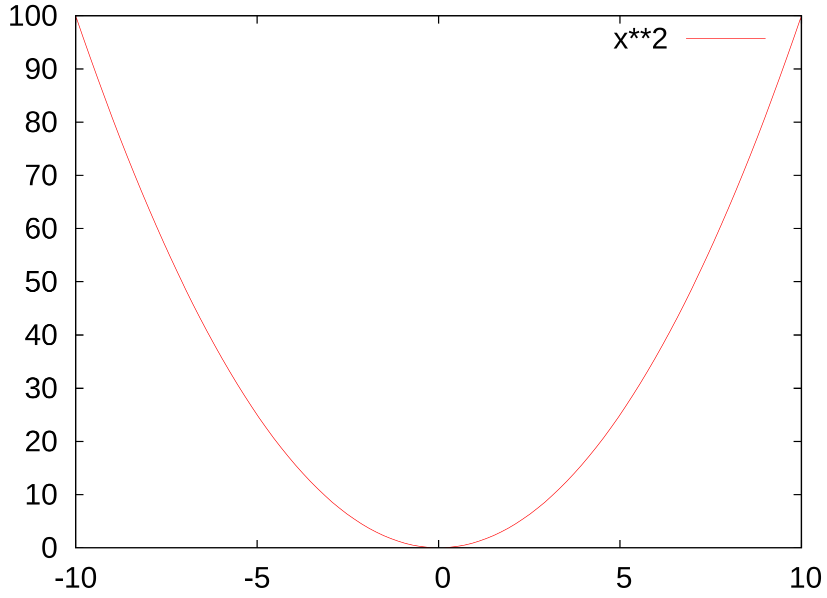
gnuplot has been around for many years and is widely used, so there are many *gnuplot* experts on the Web, offering useful advice. You'll find many *gnuplot* demos on the Web. Here's a trio of particularly informative sites:

- http://gnuplot.sourceforge.net/demo/
- http://www.gnuplotting.org/
- http://www.gnuplot.info/screenshots/

D.2. Plotting functions:

Plotting 2-d functions in *gnuplot* is quite intuitive for most people. In the example below, we're plotting a parabola. (In *gnuplot* ** means ˝exponentiate˝.)

```
plot x**2
```



By default, *gnuplot* displays 2-d functions with lines and 3-d functions with a mesh surface. The next few examples show how we can control the style with which functions are displayed. (See Figure D.1.)

Plotting a symbol at each point:

```
plot x**2 with points
```

Explicitly connecting the points with lines (this is the default):

```
plot x**2 with lines
```

Displaying a symbol at each point, AND connecting the points with lines:

```
plot x**2 with linespoints
```

Displaying an ˝impulse˝ (a narrow vertical line) for each point:

```
plot x**2 with impulses
```

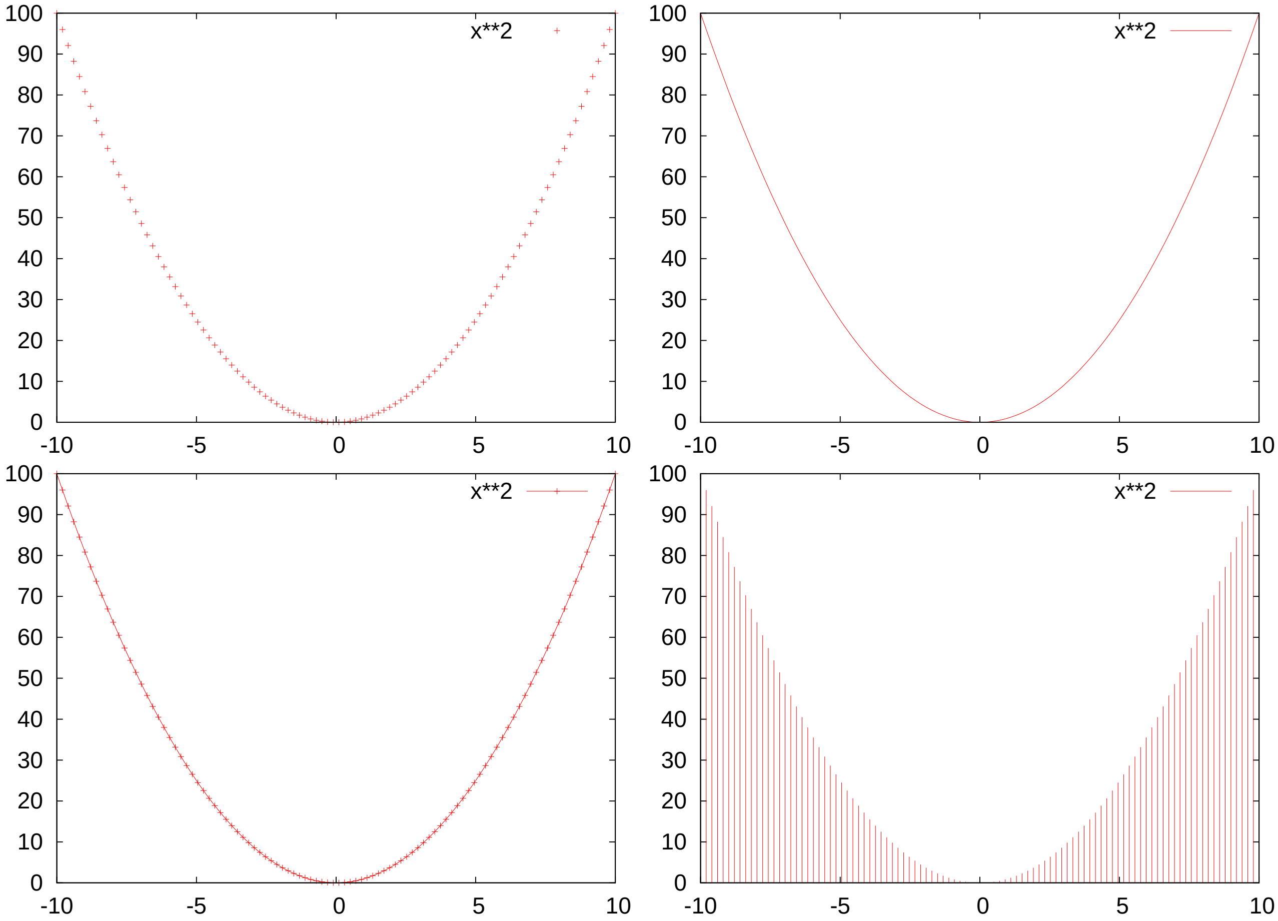


Figure D.1: Top row, left to right: plot with `points` and `lines`. Bottom row, left to right: plot with `linespoints` and `impulses`.

Displaying a box for each point (like a histogram). (Note that *gnuplot* doesn't have much of a built-in ability to generate histograms from data, but I'll show you later how you can fool it into making passable histograms without too much trouble.)

```
plot x**2 with boxes
```

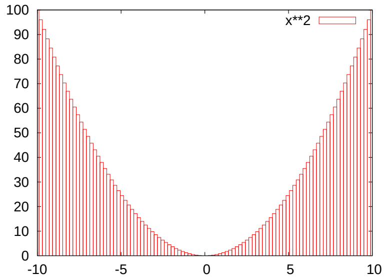


Figure D.2: A plot using the `boxes` style.

Here's our first look at a 3-d function. Note that, if you display this in *gnuplot*, you can grab the graph and move it around in three dimensions, to display it from different angles.

```
splot x**2+y**2
```

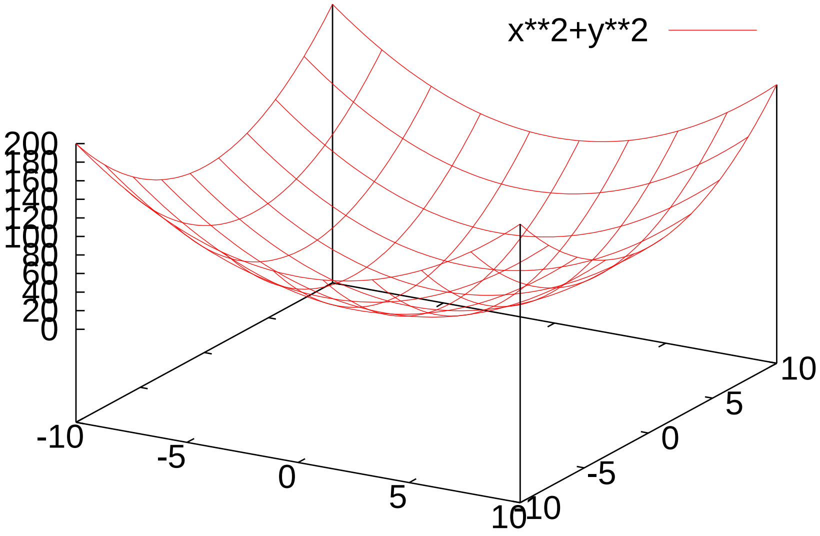


Figure D.3: A function of two variables, plotted using `splot` (for "Surface Plot").

When *gnuplot* plots a function, it generates a set of points within a range of X values (or X and Y values, for 3-d functions), then displays those points. Later, we'll see how to control the number of points. By default, *gnuplot* selects X, Y (and Z, if applicable) ranges based on some internal algorithms that generally do a pretty good job of showing the function's interesting features. We can also explicitly tell *gnuplot* what these ranges should be, as we'll see later.

D.3. Defining Functions:

As I mentioned, gnuplot has many built-in functions. Here's a plot of the sine of x:

```
plot sin(x)
```

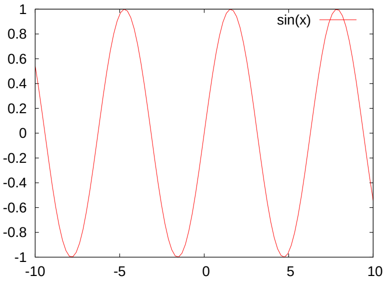


Figure D.4: The built-in function `sin(x)`.

You can also define your own functions, perhaps using some of *gnuplot*s functions as building-blocks:

```
f(x) = sin(x)*exp(x/(2.0*pi))
plot f(x)
```

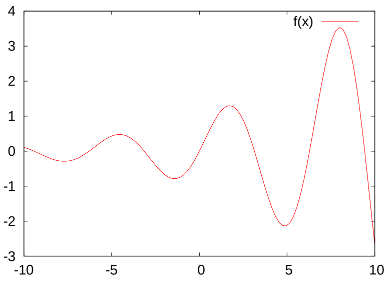


Figure D.5: A plot of $sin(x)e^{\frac{x}{2\pi}}$.

Note that *gnuplot* predefines ˝pi˝ for us. You can define your own variables, too. In the following example, we define a function of x. The function uses a parameter ˝s˝, which we can set to whatever value we want:

```
s = 10.0;
f(x) = exp(-x**2/(2*s**2))
plot f(x)
```

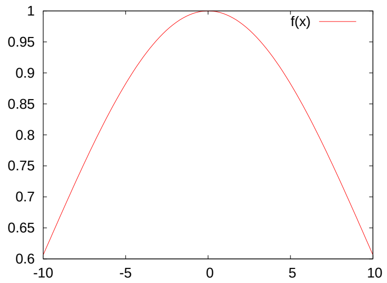


Figure D.6: A gaussian curve with s=10.

Now we can change the value of s, and plot the function again:

```
s = 1.0
plot f(x)
```

Note that *gnuplot* lets you repeat the last graphing operation by just typing ˝replot˝:

```
s = 2.0
replot #<-- note
```

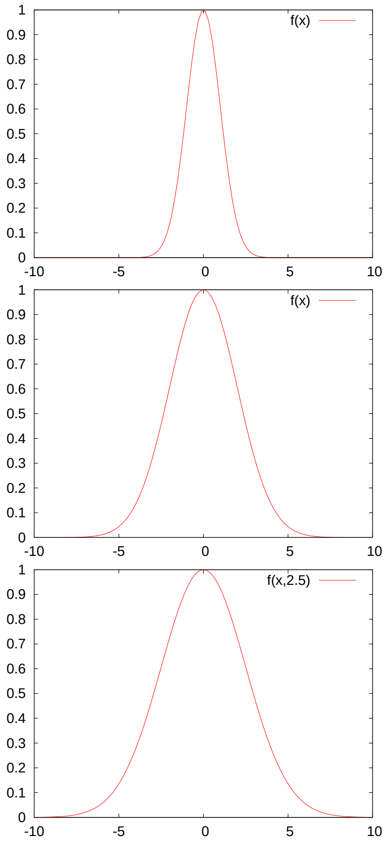


Figure D.7: Top to bottom: Gaussian curves with s = 1, 2, and 2.5.

And also note, above, that you can insert comments anywhere on the *gnuplot* command line by preceding them with a ˝#˝. This will be useful when you start writing scripts for *gnuplot*.

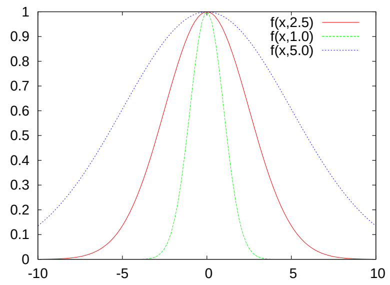
We can see the current value of a variable by using the ˝print˝ command, and we can erase a variable completely by using the ˝undefine˝ command:

```
print s
undefine s
```

Here's another way we could have defined the function f(x) above. Here we pass the parameter explicitly as one of the function's arguments:

```
f(x,s) = exp(-x**2/(2*s**2))
plot f(x,2.5)
```

Now we can easily plot a family of curves with different values of this parameter. In gnuplot, you can plot many different things with a single plot command. The things you want to plot are separated by commas. By default, *gnuplot* will try to automatically set the displayed ranges so that everything fits on the graph.



```
plot f(x,2.5), f(x,1.0), f(x,5.0)
```

Here's another way of displaying functions. The ˝filledcurves˝ style takes several parameters. In this example, we give it the parameter ˝y1=0˝, which says to fill the area between the curve and y=0. (We use ˝y1˝ because *gnuplot* allows several different y axes – one at left and one at right, for example. ˝y1˝ is the first y axis.)

```
plot besj0(x) with filledcurves y1=0, besj1(x) with filledcurves y1=0
```

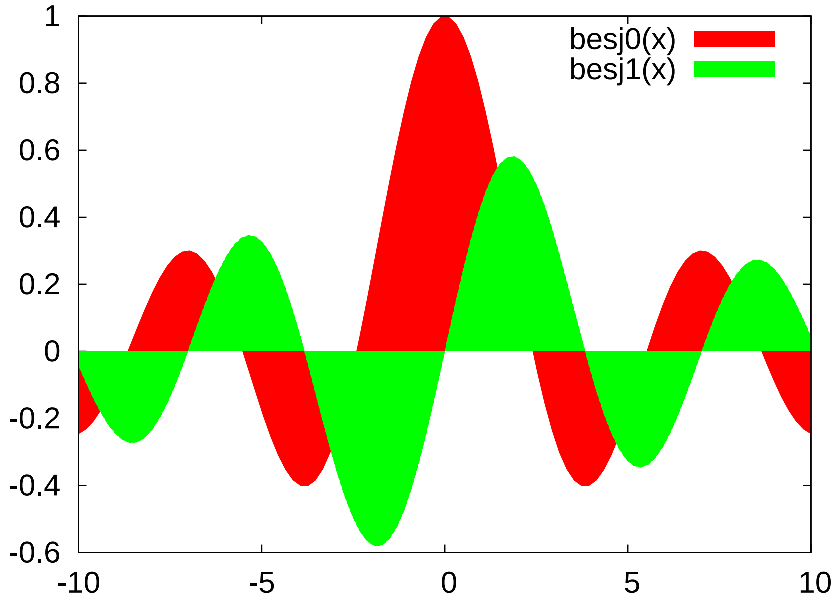
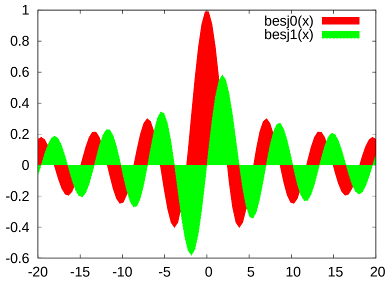


Figure D.8: Bessel functions plotted with filled curves.

D.4. Setting Ranges:

Until now we've let *gnuplot* decide what ranges of X and Y values to display. Here's how we can tell *gnuplot* to display an explicit range:



```
set xrange [-20:20]
replot
```

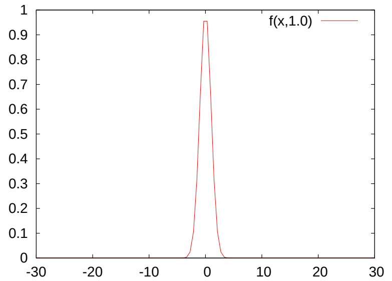
Once set, this range is used for all subsequent plots. We can also set a one-time range right along with the ˝plot˝ command:

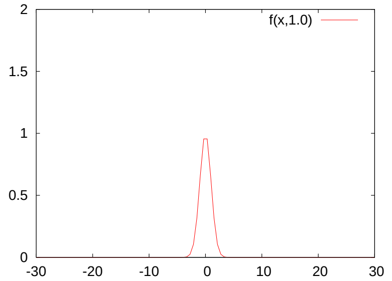
```
plot [-30:30] f(x,1.0)
```

A range set as in the example above will ony affect the current plot.

Just as with the X range, we can of course set the Y range (and the Z range, when appropriate):

```
set yrange [0:2]
replot
```





You can view the current ranges by typing ˝show xrange˝ or ˝show yrange˝ or ˝show zrange˝. You can reset the a range to auto-scaling by giving the range the value ˝[*:*]˝:

```
set xrange [*:*]
set yrange [*:*]
```

D.5. Multiple Plots:

gnuplot lets you display multiple plots on a single page. To do this, use the ˝set multiplot˝ command. Here are some examples:

```
set multiplot layout 1,2
plot f(x,1)
plot f(x,5)
unset multiplot
```

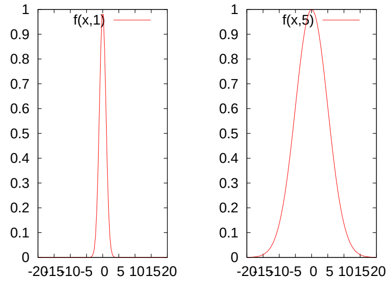


Figure D.9: `layout 1,2` creates two side-by-side plots.

Choosing `layout 1,2` creates two side-by-side regions for plotting. If we choose `layout 2,1` we get two regions, one on top of the other.

```
set multiplot layout 2,1
plot f(x,1)
plot f(x,5)
unset multiplot
```

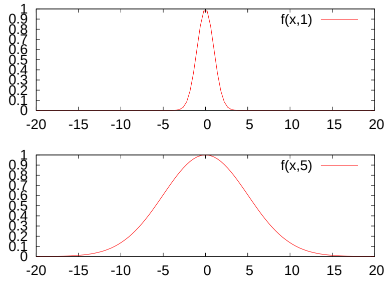


Figure D.10: `layout 2,1` creates two vertically-stacked plots.

As you might have guessed, the two numbers after `layout` just tell *gnuplot* how many horizontal and vertical regions the display should be divided into. If we want to display four plots in a 2×2 grid, we can do this:

```
set multiplot layout 2,2
plot f(x,1)
plot f(x,5)
splot x**2+y**2
splot x**3+y**3
unset multiplot
```

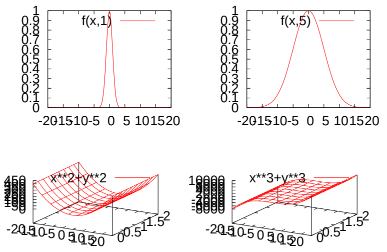


Figure D.11: A 2×2 grid of four plots.

D.6. Keys, Titles, and Labels:

You may have noticed that *gnuplot* places a ″key″ in the upper right-hand corner of each plot, identifying the information that's being plotted. You may sometimes want to turn this off. *gnuplot* provides a mechanism for this:

```
unset key
replot
```

To turn it back on, you can use the following:

```
set key
replot
```

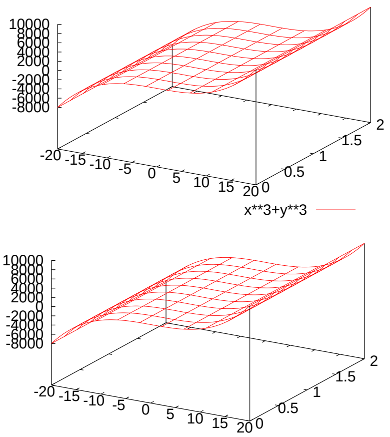


Figure D.12: Plots with (bottom) and without (top) a "key".

You can control the labels on the key by using the ″title″ option of the plot command. For example:

```
plot f(x,2) title "sigma=2",f(x,3) title "sigma=3"
```

We can also set a global title for the graph, as follows:

```
set title "some examples"
```

Axes can be labeled by using ″set xlabel″ or ″set ylabel″:

```
set xlabel "This is the x axis"
set ylabel "This is the y axis"
replot
```

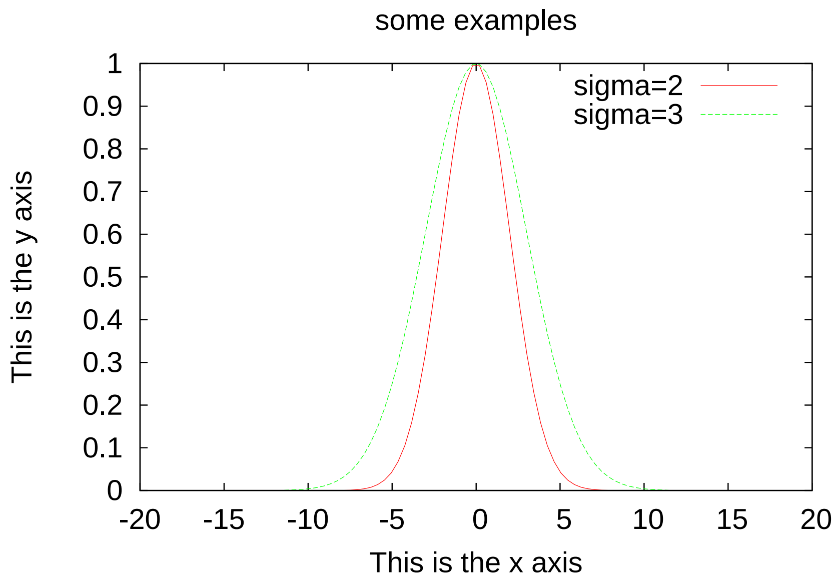
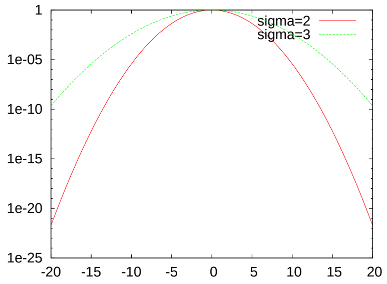


Figure D.13: A plot showing key titles, a global title, xlabel, and ylabel.

D.7. Linear and Logarithmic Scales:

Until now, we've only looked at linear scales. You might sometimes want logarithmic scales, instead. The following command makes the Y axis logarithmic:

```
set log y
replot
```

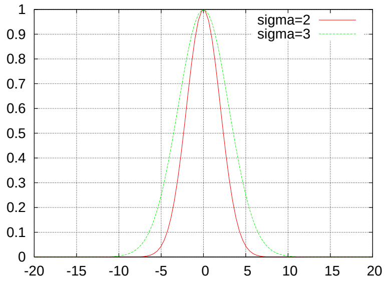


You can use ˝unset log y˝ to go back to a linear scale. You can also set log/linear scales on the X and Z axes.

```
unset log y
```

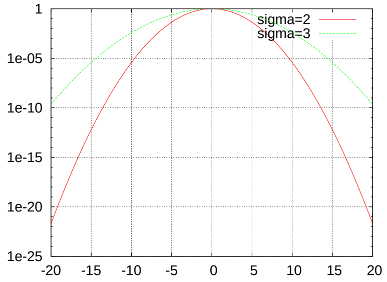
Grids are often useful for reading data off of graphs. Use the ˝set grid˝ command to turn on a coarse-grained grid on your graph:

```
set grid
replot
```



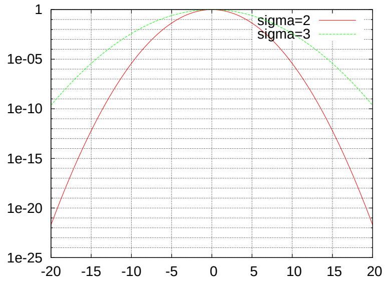
With logarithmic scales, these coarse-grained grid lines will often be unsatisfactory:

```
set log y
replot
```



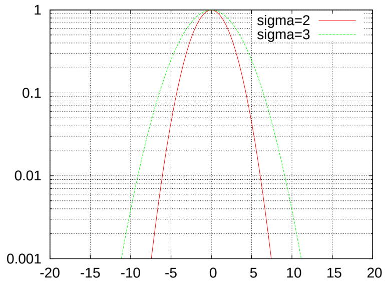
In this case, we may want to turn on ˝minor˝ grid lines to. To do this we use some of the available qualifiers for the ˝set grid˝ command. ˝ytics˝ here refers to the major tic marks on the Y axis. ˝mytics˝ refers to the minor tick marks. The command below tells *gnuplot* to make grid lines for both major and minor tic marks.

```
set grid ytics mytics
replot
```



You can see that *gnuplot* doesn't always choose reasonable ranges for the axes, especially when the axis is logarithmic. We can make this look better by explicitly setting the lower end of the range:

```
set yrange [.001:*]
replot
```



D.8. Three-Dimensional Plots:

Now let's look at some more 3-d plots. Let's start by defining a function 3-d version of the f(x) we were using above:

```
f(x,y,s) = exp(-(x**2+y**2)/(2*s**2))
set xrange [-10:10]
set yrange [-10:10]
splot f(x,y,1)
```

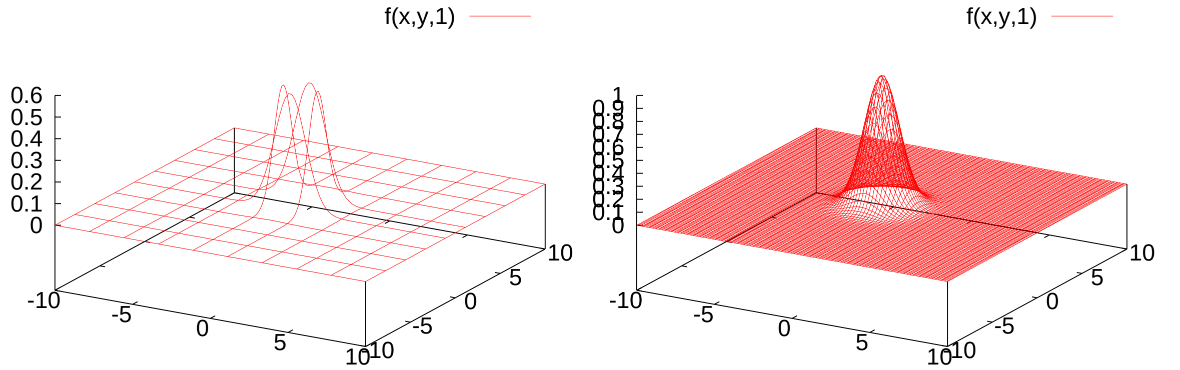


Figure D.14: 3-d plots showing the effect of `samples` and `isosamples`.

The output is shown on the left-hand side of Figure D.14. The graph looks confusing because *gnuplot* didn't evaluate the function at very many points, and didn't draw many lines in the mesh that indicates the location of the surface.

We can get a better plot by telling *gnuplot* explicitly how many points to use when sampling the function, and how many lines to draw across the surface. The first of these is controlled by *gnuplot*'s ″samples″ setting, and the second by the ″isosamples″ setting. As you can see this makes the graph much better, as shown in the right-hand side of Figure D.14.

```
set samples 100
set isosamples 100
replot
```

But why is the zero of the Z axis lifted up like that? This is so *gnuplot* can display a contour map underneath, as we'll see later. For now, if we don't like the Z offset we can eliminate it:

```
set xyplane 0
replot
```

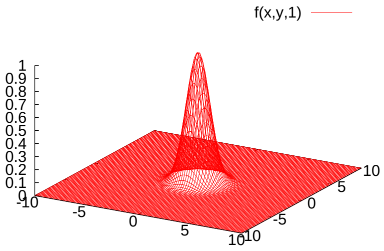
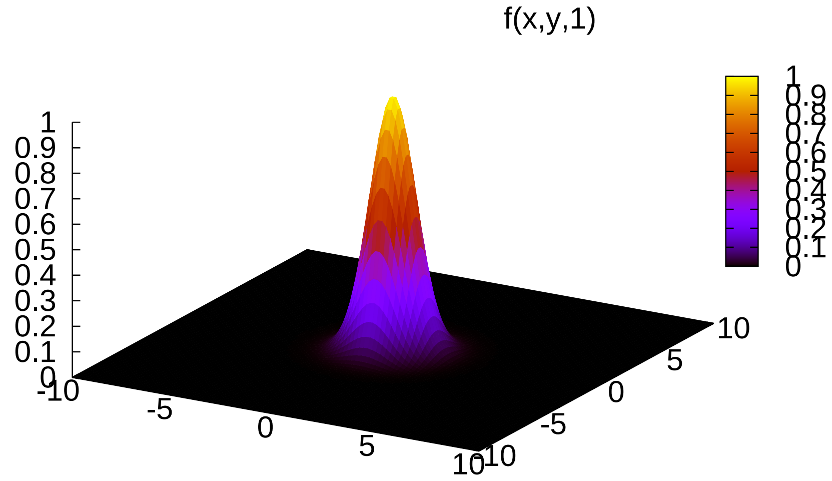


Figure D.15: The effect of setting `xyplane` to zero.

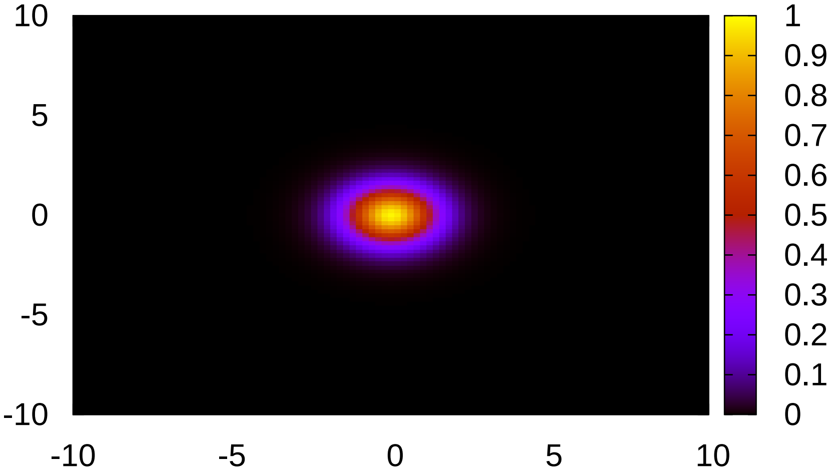
gnuplot provides other ways of displaying 3-d data. One of these is called ˝pm3d˝. This style colorizes the surface based on the Z-value at each point. Here's an example:

```
splot f(x,y,1) with pm3d
```



Sometimes we just want the colorization, without the 3-d look. For this, *gnuplot* provides the ˝map˝ view. This displays the data in the X-Y plane, with colors providing information about the Z values. Here's an example of that:

```
set view map
replot
```



D.9. Color Palettes:

The graphs above use a default palette of colors, but we can define our own palette if we want to. Using the ˝set palette˝ command, we can tie certain colors to certain Z values. *gnuplot* will interpolate between the colors we specify and generate a color for each Z value on the graph. We can specify as many Z values as we want to in the ˝set palette˝ command. In the example below, I specify the color for 0 and for 1, and let *gnuplot* figure out the rest. We could specify the colors at other locations by just adding more comma-separated pairs to our list:

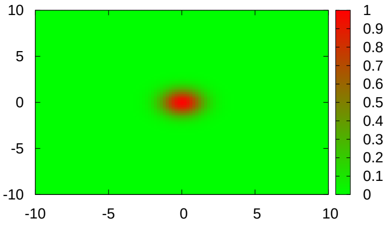


Figure D.16: A green-to-red palette.

```
set palette defined (0 "green", 1 "red")
replot
```

You can reset the palette to the default values by just typing ˝set palette˝ by itself:

```
set palette #<-- reset
replot
```

If we wanted to display a grid on a plot like this, we'd need to be careful about the color of the grid lines. By default, these lines are black, and wouldn't show up. We can specify the line color at the ˝set grid˝ line, though. Here's an example where I set the grid line color to white. Notice that I also use the ˝front˝ qualifier, to make sure the grid lines are displayed in front of the data. That's important in this case, because grid lines are normally displayed behind the data, and would be obscured by the solid colors of our dataset.

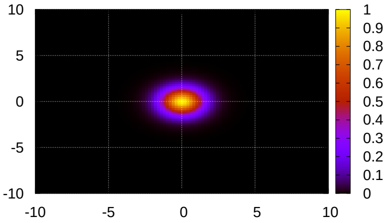


Figure D.17: Overlaying a grid.

```
set grid front xtics ytics lc rgb '#ffffff'
replot
```

Here are some examples of other built-in color palettes:

```
set palette gray
replot
```

```
set palette color negative
replot
```

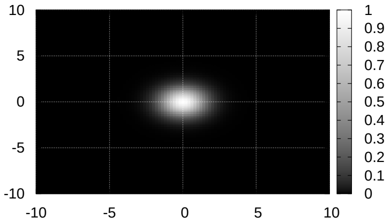


Figure D.18: A grayscale palette.

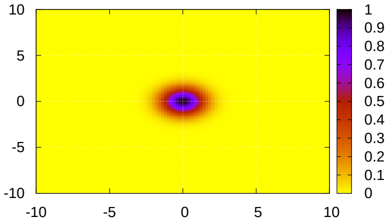


Figure D.19: An inverted-color palette.

D.10. Setting the Viewing Angle:

As we noted before, when *gnuplot* is showing us a 3-d plot it allows us to grab the plot and turn it around to view it from different angles. We can also control the viewing angles from the command line, using commands like the following. (These are actually the default values. Unfortunately, *gnuplot* doesn't provide us with a way to just ˝set view default˝. We have to explicitly enter the values.)

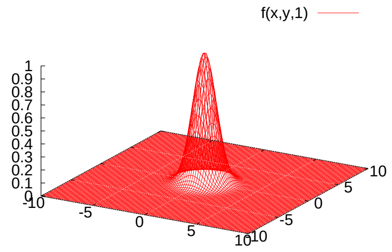


Figure D.20: A 3-d graph, rotated to specific angles using `set view`.

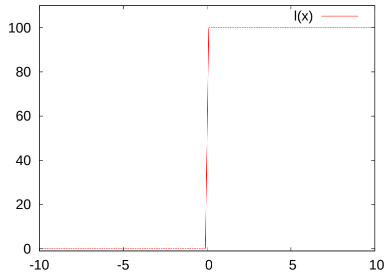
```
set view 60, 30, 1, 1 #<-- rot_x, rot_z, scale, scale_z
splot f(x,y,1)
```

D.11. Discontinuous Functions:

What if we want to plot a function that has a discontinuity? Say, a step-function? *gnuplot* makes it easy to do that, too. The following example shows one way to do it, using the ˝ternary operator˝ (?:). If you're familiar with C or Perl, you probably already know how this operator works. The syntax is ˝test ? true : false˝. If ˝test˝ is true, then the ˝true˝ section is used. Otherwise, the ˝false˝ section is used. It's like a compact if/else statement.

In this example, we say that the function l(x) has the value 100 if x is greater than 0, or a value of 0 otherwise.

```
set yrange [-1:110]
l(x) = x>0 ? 100 : 0
plot l(x)
```

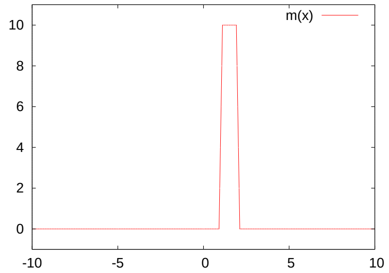


What if we wanted to define a ˝square pulse˝, i.e., a function that only has a non-zero value between x=x1 and x=x2? We could do that by first defining a gneralized step function:

```
l(x,x0,a) = x<x0 ? 0 : a
```

In the function above, `x0` is the x value at which the function changes value, and `a` is the value it has when it's non-zero. Now we can construct a square pulse by taking the difference of two instances of this function with different `x0` values:

```
m(x) = l(x,1,10) - l(x,2,10)
set yrange [-1:11]
plot m(x)
```



(Notice that the ˝vertical˝ lines aren't exactly vertical. That's because *gnuplot* is just connecting a discrete set of data points it has generated along the function. We could improve the plot by using ″set samples″ to increase the number of data points.)

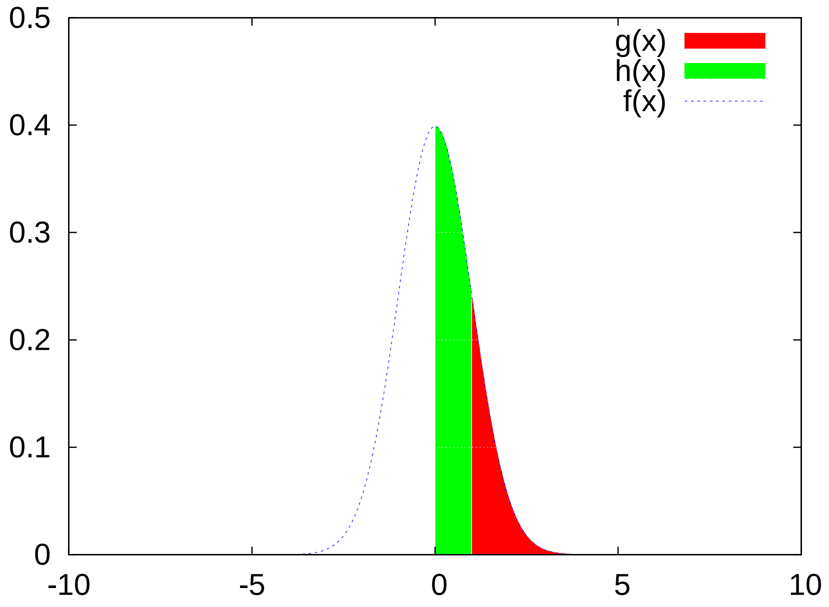
I think it's clear that we can construct any arbitrarily complex disjoint function by using similar mechanisms.

D.12. Hiding Regions:

Sometimes we want *gnuplot* to just display nothing in certain regions. Perhaps the function is undefined there, or maybe we just want to emphasize a certain region. Here's a trick to make that happen. Can you figure out how it works?

(Also notice that the example below uses ″filledcurves x1″ to cause some areas to be filled between the curve and the bottom of the graph.)

```
# Other piecewise functions, using sqrt(-1) to make function disappear:
set samples 1000
set yrange [0:0.5]
f(x) = exp(-x*x/2)/sqrt(2*pi)
g(x) = x>=1 ? f(x) : sqrt(-1)
h(x) = x<=1 && x>=0 ? f(x) : sqrt(-1)
plot  g(x) with filledcurves x1,h(x) with filledcurves x1, f(x) with lines
```

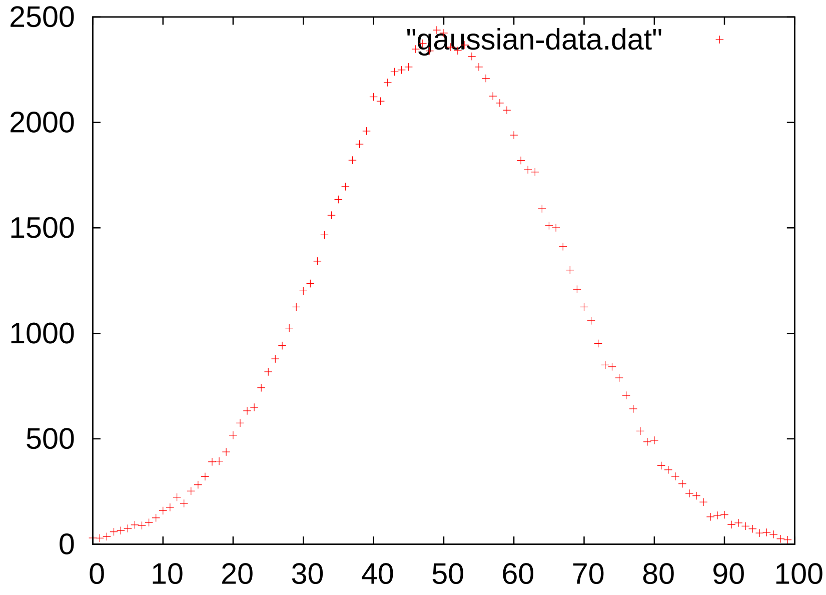


D.13. Plotting Data:

OK, so we've seen how gnuplot works for plotting functions. How about plotting data points? We can use the same tools we've seen above for controlling the look of the graph, no matter whether we're plotting functions or data. We can also still use the ˝plot˝ and ˝splot˝ commands.

Here's a simple example showing how to use *gnuplot* to plot data from a text file. The file contains three columns of numbers, separated by white space. In this example, the colums are, in order, X, Y and the error in Y.

```
  plot "gaussian-data.dat"
```



The command above just reads the the first two columns and plots the data as X and Y values, placing a symbol at each point.

Note that the file name must always be enclosed in quotes.

We can tell *gnuplot* to make use of the ˝error˝ column by adding ˝with errorbars˝:

```
  plot "gaussian-data.dat" with errorbars
```

gnuplot will assume that the third column in the file contains the error values, unless we tell it otherwise.

We can also explicitly tell *gnuplot* which columns to use for X, Y, error values, and so forth. In the following example, we tell *gnuplot* to plot data from a text file, and use column 2 as the X value and column 3 as the Y value:

```
plot "h_200.dat" using 2:3
```

When plotting error bars, we can also specify a third column containing those:

```
plot "h_200.dat" using 2:3:4 with errorbars
```

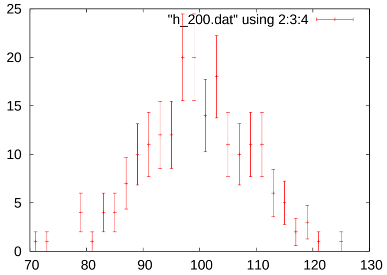
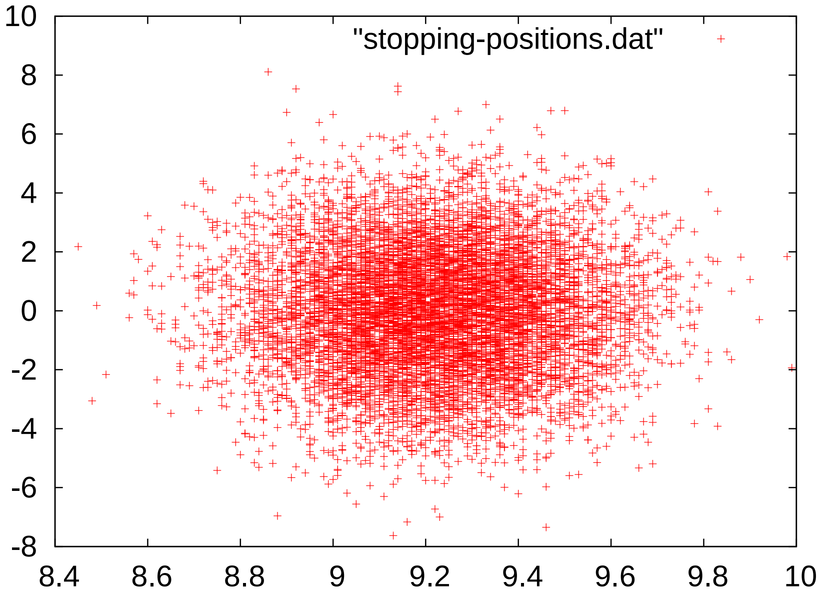


Figure D.21: Plotting selected columns with error bars.

We can also plot 3-dimensional data. Here's another plot, showing stopping positions of charged particles in a chunk of matter. The file contains three columns, representing the X, Y and Z components of the stopping position.

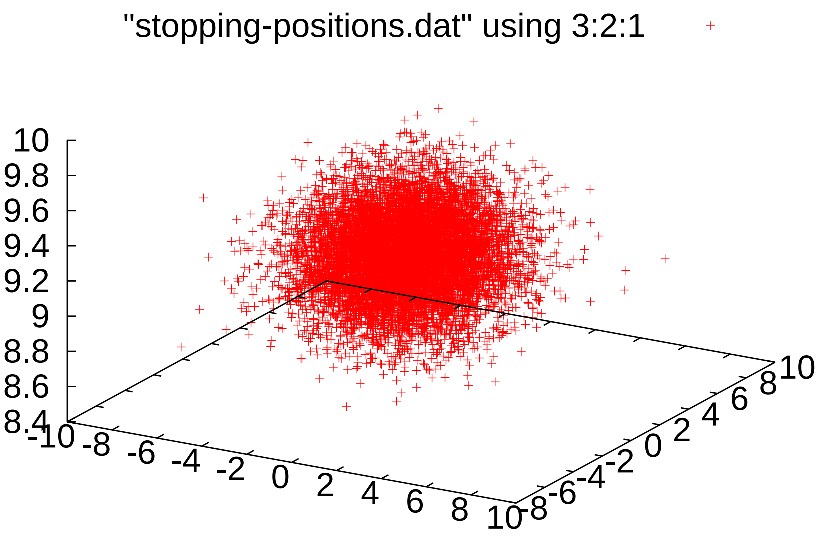
```
plot "stopping-positions.dat"
```



This is equivalent to ˝using 1:2˝.

For displaying all three dimensions we can use the same data file with *gnuplot's* `splot` command. The default order of the columns is X, Y, Z, but we can reorder them if we want. Here's a 3-d plot of the same data, using column 3 as X, column 2 as Y and column 1 as Z:

```
splot "stopping-positions.dat" using 3:2:1
```



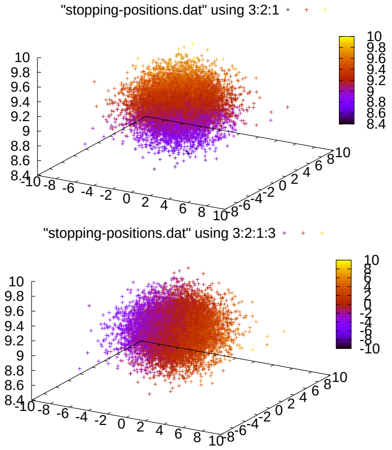


Figure D.22: Colorizing by Z-value (top) or a specified column (bottom).

We can also tell *gnuplot* to colorize the points, using the option ˝with points palette˝. (See Figure D.22.) By default, points are colorized based on the value of Z.

```
splot "stopping-positions.dat" using 3:2:1 with points palette
```

If we want to, we can specify another column to use for colorizing the points:

```
splot "stopping-positions.dat" using 3:2:1:3 with points palette
```

If we have data that we want to display in the style of a histogram (see Figure D.23), we might use the option ˝with boxes˝:

```
plot "energy.dat" using 1:3 with boxes
```

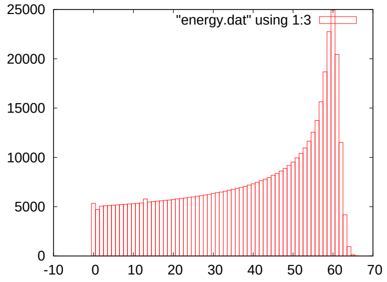
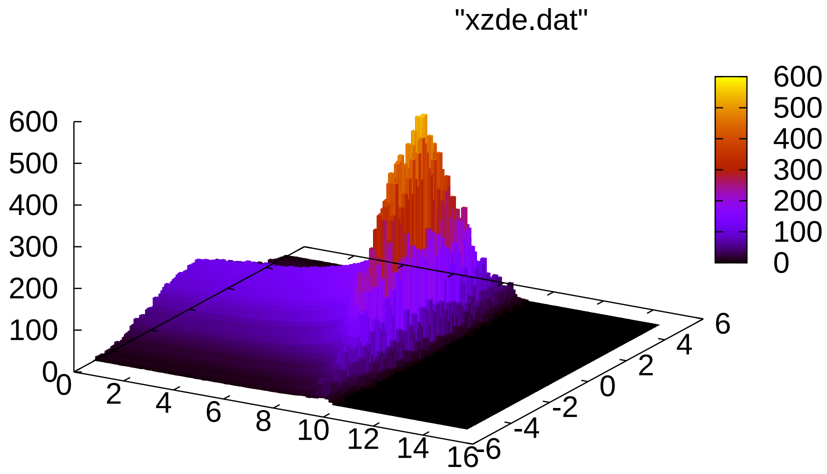


Figure D.23: Data displayed with boxes, in the style of a histogram.

Here's another data set from these stopped particles. This one contains two-dimensional histogram data, binned by X and Z, with the histogram height being the amount of energy deposited in each X,Z bin. In this case, let's use the ″pm3d″ style to colorize the graph based on the energy value:

```
splot "xzde.dat" with pm3d
```



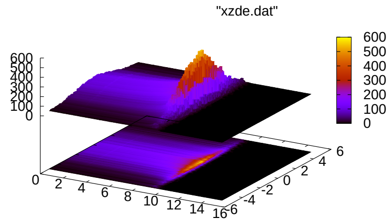


Figure D.24: Projecting the data into a color map on a plane beneath the surface.

We can tell *gnuplot* to also display a color map on the bottom of the graph. To make this visible, we'll need to lift the surface up a little. The additional ″at bs″ tells the pm3d style to colorize both the surface (″s″) and the bottom (″b″).

```
set xyplane 1
splot "xzde.dat" with pm3d at bs
```

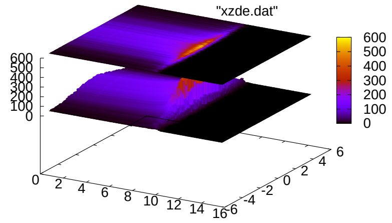


Figure D.25: Projecting the data into a color map on a plane *above* the surface.

We could place the color map at the top, instead, by saying ″at st″, for ″surface″ and ″top″. Note that the order matters, since it controls the order in which the two maps will be drawn, and one map may obscure the other if we do them in the wrong order (try it and see).

```
splot "xzde.dat" with pm3d at st
```

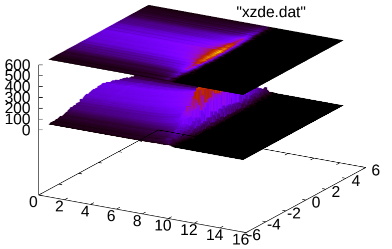


Figure D.26: The effect of `unset colorbox`.

When plotting colorized graphs, we can control whether or not we display the color key by typing ″unset colorbox″ or ″set colorbox″:

```
unset colorbox
replot
```

In the following example, we ask *gnuplot* to create a color map on the bottom surface, and also to plot a wire-mesh (the default) surface above this:

```
splot "xzde.dat" with pm3d at b, "xzde.dat" with lines
```

Sometimes we may just want to see the colormap. As we saw above, we can get this by typing ˝set view map˝. The graph below is colorized according to how much energy was deposited at each location. We can see the particles coming in from the left, depositing more and more of their energy as they slow down and stop.

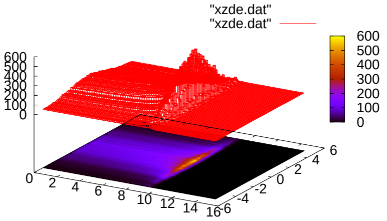


Figure D.27: A color map on the bottom, and a wire mesh on top.

```
set view map
splot "xzde.dat" with pm3d
```

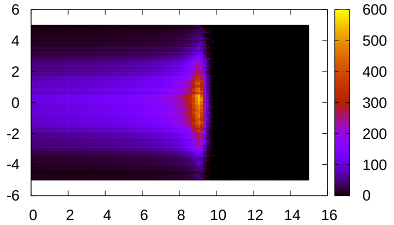


Figure D.28: The effect of `set view map`.

I mentioned above that *gnuplot* doesn't know about histograms, and can't automatically bin data for you. It's pretty straightforward to construct a simple histogram using *gnuplot*'s functions, though. Here's an example, using the X value of the particle stopping position data. In the following, I define a function, ˝bin(x)˝, which just returns the X value of the center of the bin into which a given data point would fall. We then make use of an ability of *gnuplot*'s to plot the sum of all Y values with the same X value.

As X values, we plot bin(x), and for each value we give *gnuplot* a fixed Y value of 1. We mean by this, ˝1 particle stopped inside the bin on the X axis˝. We then tell gnuplot to use ˝smooth freq˝, which is a style that causes *gnuplot* to sum all of the Y values at a given X value, and display the result. We've created a histogram! (We'll talk more about the syntax of this ˝using˝ statement later.)

Here's what it looks like:

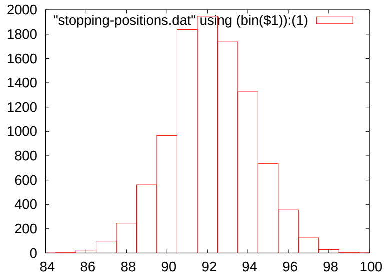


Figure D.29: A sneaky histogram.

```
# A sneaky way to make histograms:
# See also:
# http://www.inference.phy.cam.ac.uk/teaching/comput/C++/examples/gnuplot/#four

binsize = 0.1
bin(x) = int( x/binsize + 0.5 )
plot "stopping-positions.dat" using (bin($1)):(1) smooth freq with boxes
```

D.14. Binary Data Files:

Up until now, we've read data from ASCII files. *gnuplot* can also read binary files. We just need to tell *gnuplot* that the file is binary, and what kind of numbers are in it. For example, the following command reads a binary file containing floating-point data (type `double` in C and *gnuplot* parlance). The file was created by a C program, which wrote the numbers in binary format into the file. Below, we tell *gnuplot* that the file contains a stream of ˝doubles˝. If the file were a different format (say, alternating `double` and `int`), we could tell *gnuplot* how to deal with it (say, `format = "%double%int"`). Type ˝help plot binary general˝ in *gnuplot* for more information.

```
# binary data
plot "data.dat" binary format="%double"
```

D.15. Mathematical Combinations of Data:

As we saw in the histogramming example above, *gnuplot* lets us plot functions of data columns. We specify what to plot with the ˝using˝ qualifier. If we're just plotting the unadorned contents of the column, we just give the column's number. But, if we want something more complicated, we can supply a more complicated expression. These more complicated expressions need to be enclosed in parentheses. Within these parentheses we can use whatever arithmetic expressions and functions we want, referring to data by column number. In this context, the column numbers must be preceded by ˝$˝, to distingush them from actual numbers that we might be using in the expressions.

Here's a pair of examples (see Figure D.30):

```
plot "stopping-positions.dat" using 1:($2/100)
```

```
plot "stopping-positions.dat" using (sqrt($1**2+$2**2+$3**2))
```

Sometimes we want to use ˝line number˝ as one of the things we plot. For example, imagine we have a file containing many measurements of position and time. Each line of the file just has two values, `x` and `t`. If the lines in the file are in the same order in which we did the measurements, we could think of the line number as a third value: the ˝measurement number˝. We can use the line number in our plots by referring to `(column(0))` or, equivalently, `($0)`. For example, to plot position versus line number:

```
plot "mydata.dat" using ($0):1
```

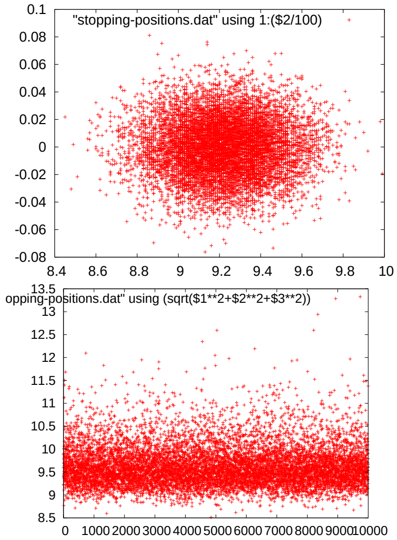
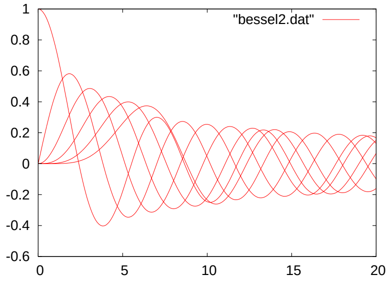


Figure D.30: Plots produced by the "using" expressions in the two examples at left.

D.16. Multiple Data Sets in One File:

A data file may contain more than one data set. In the example below, we plot data from a file called ˝bessel2.dat˝ which contains five data sets. Each data set is two columns containing x and jn(x), where jn is the nth order Bessel function. The first data set is j0(x), the second is j1(x) and so on. The data sets are just concatenated together, with blank lines separating them.

```
# Multple data sets in one file, with blank lines:
set xrange [0:20]
set yrange [*:*]
plot "bessel2.dat" with lines
```

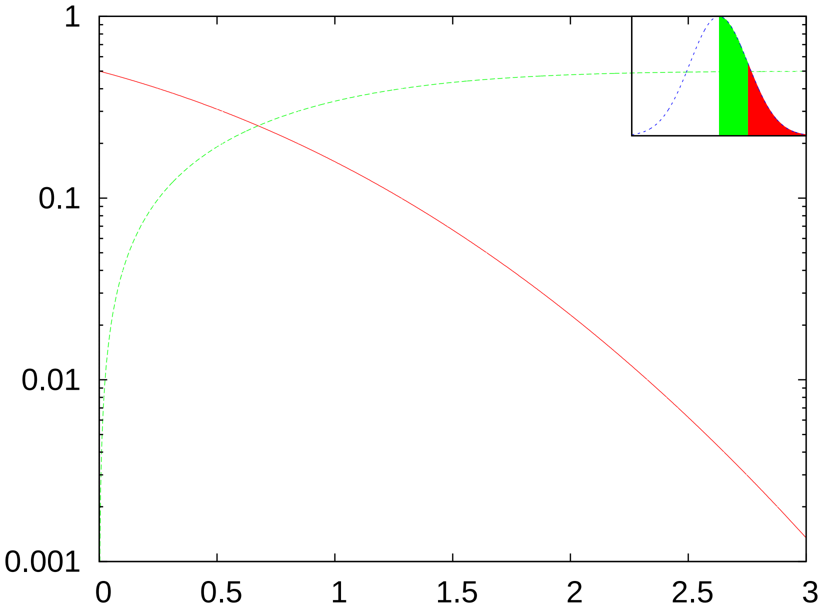


D.17. Inset Graphs:

Sometimes we want to have a smaller graph inset into a larger one. Here's a long example that illustrates how to accomplish that in *gnuplot*. Within the ˝multiplot˝ environment, we can specify the size and location of each plot explicitly. In the example below, we create a large graph by specifying ˝origin 0.0,0.0˝ and ˝size 1.0,1.0˝. Multiplot's coordinate system (by default) begins at 0,0 in the lower left corner of the screen and goes to 1,1 at the upper right. We then set the origin and size of a second plot so as to place it in the upper right corner of the first graph.

```
i(x) = 0.5*(1+erf(x/sqrt(2)))
unset key
unset label
unset xlabel
unset ylabel
unset title

set multiplot

set origin 0.0,0.0
set size 1.0,1.0

set yrange [0.001:]
set xrange [0:3]
set log y
set xtics
set ytics
set grid xtics ytics mxtics mytics
```

```
plot 1-i(x), 0.5-(1-i(x))

set origin 0.7,0.7
set size 0.3,0.3

f(x) = exp(-x*x/2)/sqrt(2*pi)
g(x) = x>=1?f(x):sqrt(-1)
h(x) = x>=1&&x>=0?f(x):sqrt(-1)
set xrange [-3:3]
unset log y
unset grid
unset xtics
unset ytics
plot  g(x) with filledcurves x1,h(x) with filledcurves x1, f(x) with lines
unset multiplot
```



D.18. Writing Output Files:

When you start *gnuplot* and begin graphing, *gnuplot* chooses one of several ways of displaying the data, depending on the abilities of your computer's display. Each way of displaying the data is called a ˝terminal type˝ or ˝term˝ in *gnuplot*. If you're using *gnuplot* under Linux, you'll probably be using the x11 or the wxt term. You'll see a message when you start *gnuplot* that says something like ˝Terminal type set to 'wxt'˝. Both of these terminal types are used for displaying graphs on your computer's screen. You can see what the current terminal is by typing ˝show term˝.

There are other terminal types that are intended for creating graphics files. For example, you can use the ˝png˝ terminal to create png files, or the ˝postscript˝ terminal to creat postscript files.

In the example below, we change the terminal type to ˝postscript˝ using the command ˝set term postscript enhanced color˝. The ˝enhanced color˝ part specifies some options available in the postscript terminal type. If we tried plotting a graph at this point, we'd see postscript commands printed on our screen. We don't want that! The next thing we need to do is to tell *gnuplot* where to write these postscript commands. We do this by using the ˝set output˝ command. Note that the name of the output file must be enclosed in quotes. Anything we subsequently plot will be written into this file as postscript data.

```
set term postscript enhanced color
set output "gnuplot/images/file.eps"
plot "energy.dat" using 1:3 with boxes
```

We can similarly send output into a png file:

```
set term png
set output "gnuplot/images/file.png"
replot
```

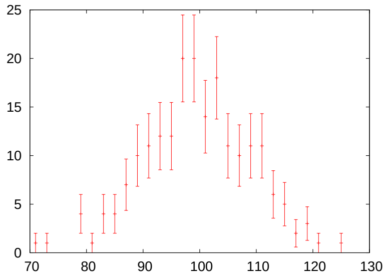
Many terminal types allow you to use special symbols (e.g., Greek letters) in titles and labels. Unfortunately, the way to do this varies greatly from one terminal type to another. For example, to produce a lower-case Greek sigma with the postscript driver, you could insert the string `{/Symbol s}` in your title. For the png terminal, you'd need to insert a unicode symbol by typing an appropriate sequence of keys on your keyboard. For one of the Latex terminal types, you'd need to use Latex-style equations. A useful cheat-sheet for this kind of thing can be found at http://mathewpeet.org/lists/symbols/.

D.19. Fitting functions to data:

gnuplot also allows us to fit model functions to data sets by searching through parameter-space to find a set of parameters that minimize the chi-squared value obtained by comparing the given model to the data set.

For example, consider the following data set, which contains some data that appears to be distributed in something like a Gaussian distribution.

```
        set xrange [*:*]
        set yrange [*:*]
        plot 'h_200.dat' using 2:3:4 with errorbars
```



We can define a function that represents a generalized Gaussian distribution, characterized by three parameters: s (the standard deviation), m (the mean) and a (an amplitude). We define such a function, g(x), below. *gnuplot* is capable of adjusting the values of a, m and s in order to find the best fit to a given data set. *gnuplot* isn't particularly good at guessing good initial values for these parameters, so we should set them by hand to some approximate values before asking *gnuplot* to adjust them. In the example below, we just read approximate values from the graph, without too much care.

Then, we use *gnuplot*'s fit command to adjust the parameters a,m and s to find a minimum chi-squared.

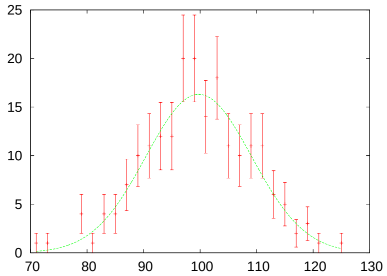


Figure D.31: Data plotted along with a best-fit curve.

```
        g(x) = a*exp(-(x-m)**2/2/s**2)  # Gaussian
        a=25
        m=100
        s=15
        fit g(x) 'h_200.dat' using 2:3:4 via a,m,s
```

The output of the fit command will look something like this:

```
After 5 iterations the fit converged.
final sum of squares of residuals : 11.2835
rel. change during last iteration : -4.51108e-07

degrees of freedom    (FIT_NDF)                        : 22
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.716162
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.512888

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 16.3064          +/- 1.084        (6.645%)
m               = 99.7578          +/- 0.5116       (0.5128%)
s               = 9.36694          +/- 0.4236       (4.522%)
```

We can then ask *gnuplot* to draw the best-fit function (using the newly-obtained parameter values), along with the data (see Figure D.31):

```
plot 'h_200.dat' using 2:3:4 with errorbars, g(x)
```

D.20. Using text as axis labels:

It's sometimes useful to be able to use text as axis labels. For example, you might have a file like this:

```
Joe 1.00
Bob 2.45
Mary 3.14
Jane 0.76
```

You could plot these values with the names as labels by typing the following in *gnuplot*:

```
plot "file.dat" using 2:xticlabels(1) with boxes
```

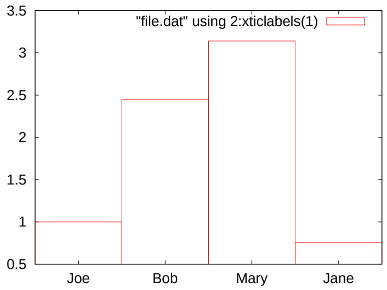


Figure D.32: Using a text column to label an axis.

If the labels are so long that they bump into each other, you can rotate them by issuing the command:

```
set xtics rotate by -90
```

If you do this, you may also need to reduce the height of the graph to leave vertical room for the labels. This can be done with a command like:

```
set size ratio 0.7
```

D.21. Using dates and times in data sets:

Finally, *gnuplot* is capable of reading date and time data in data files, and plotting them appropriately. For detailed information, type ″help set xdata″ and ″help set timefmt″ inside *gnuplot*. One quick example is shown below. In it, we tell *gnuplot* that the X values will be times, and that their format in the data file will be abbreviated month names (like ″Jan″, ″Feb″, etc.) Then we tell *gnuplot* to mark the X axis with labels in the same format. After that, we only need to tell *gnuplot* to plot the data in the file.

```
  set xdata time     # Tell gnuplot that the X values will be times.
  set timefmt "%b"   # Tell gnuplot what format to expect in the data file.
                     # See man strftime for codes.
  set format x "%b"  # How axis will be displayed.
  plot "mail-stats.dat" using 1:2 with boxes
```

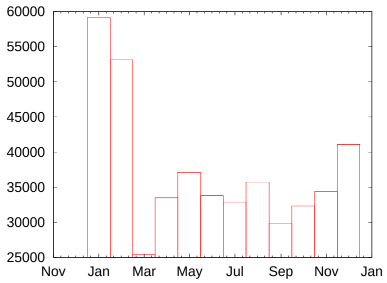


Figure D.33: Using dates or times to label an axis.

E. Format Specifier Tricks

Here are a few tips and tricks for format specifiers (placeholders) in `printf` and `scanf` statements. This isn't an exhaustive list of the things you can do with format specifiers, but it includes the things you're likely to use most often. For complete information, see the excellent Wikipedia article on "printf format strings".

To start with, here's a list of format specifiers for some common variable types:

Format	Description
`%d`	Format for printing or reading an integer.
`%lf`	Format for printing or reading a `double`.
`%le`	Print a `double` in scientific notation.
`%lg`	Print a `double` in either scientific notation or normal notation, whichever is more appropriate.
`%c`	A single character.
`%s`	An array of characters (also called a "character string")
`%p`	A memory address (also called a "pointer").
`%u`	An unsigned integer.
`%ld`	A long integer.
`%lu`	An unsigned long integer
`%lld`	A long long integer.
`%llu`	An unsigned long long integer.
`%x`	A hexadecimal integer with lower-case letters.
`%X`	A hexadecimal integer with upper-case letters.
`%o`	An octal integer.

You can adjust the behavior of these format specifiers by adding modifiers to them. The following tables shows some tricks to help you make your program's output look just the way you want it.

Generic tricks		
Example	Result	Description
printf("%%")	%	Print a literal % symbol.

Tricks for Integers		
Example	Result	Description
printf("%**20**d",1234567890)	1234567890	Print an integer, reserving a 20-digit-wide space for it. If the number isn't this long, add blank spaces on the left-hand side.
printf("%**-20**d",1234567890)	1234567890	The same as above, but add blank spaces on the *right*-hand side if necessary.
printf("%**8**d",1234567890)	1234567890	If the number won't fit in the specified width, use as much space as necessary.
printf("%**020**d",1234567890)	00000000001234567890	Pad the number with zeros on the left (if necessary) to make it 20 digits long.

Tricks for doubles		
Example	Result	Description
printf("%**20**lf",M_PI)	3.141593	Print a double, reserving enough space for 20 digits.
printf("%**5**lf",M_PI*1e8)	314159265.358979	If the number won't fit in the specified width, use as much space as necessary.
printf("%**20.10**lf",M_PI)	3.1415926536	Reserve enough space for 20 digits (including the decimal point) and print 10 of those digits after the decimal point. Pad with spaces on the left-hand side if necessary.
printf("%**-20**lf",M_PI)	3.141593	As above, but pad on the *right*-hand side.
printf("%**020**lf",M_PI)	0000000000003.141593	Pad the number with zeros on the left (if necessary) to make it 20 digits long.

Tricks for Characters		
Example	Result	Description
printf("%**20**c",'A')	`                   A`	Print a character, reserving a 20-character-wide space for it. Fill the extra width with spaces on the left-hand side.
printf("%**-20**c",'A')	`A                   `	As above, but fill on the *right*-hand side.

Tricks for Strings		
Example	Result	Description
printf("%**20**s","Testing")	`             Testing`	Print a character string, reserving a 20-character-wide space for it. Fill the extra width, if any, with spaces on the left-hand side.
printf("%**-20**s","Testing")	`Testing             `	As above, but fill with spaces on the *right*-hand side.
printf("%**4**s","Testing")	`Testing`	If the string won't fit in the specified width, use as much space as necessary.