4. *Math and More Loops*

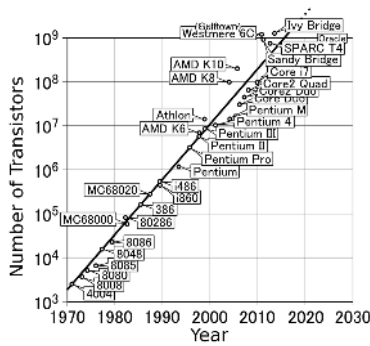


Figure 4.1: An illustration of "Moore's Law" for CPUs. Note that the vertical axis is logarithmic.
Source: Wikimedia Commons

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.

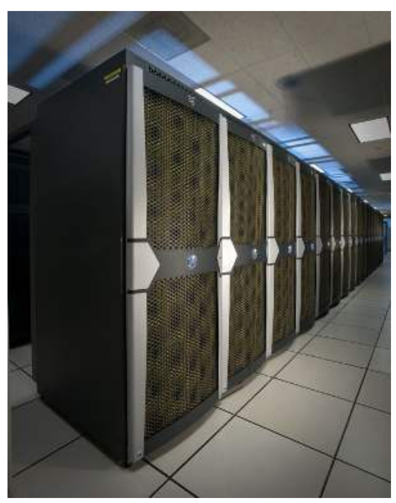
[1] Moore, G. E. Electronics 38, 114-117 (1965).

[2] Nature 530, 144-147 (11 February 2016).

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.



A modern supercomputer: NASA's *Pleiades* Cluster.
Source: Wikimedia Commons

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.

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:

$$\begin{aligned} \texttt{2.5e+3} &= 2,500 \\ \texttt{6.02e+23} &= 6.02 \times 10^{23} \text{ (} \simeq \text{Avogadro's number)} \\ \texttt{5e-11} &= 5 \times 10^{-11} \end{aligned}$$

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.

[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.

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]

[4] Some compilers will automatically set all variables to zero at the beginning of a program, but it's best not to assume this.

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 );
}
```

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.

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

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?

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.

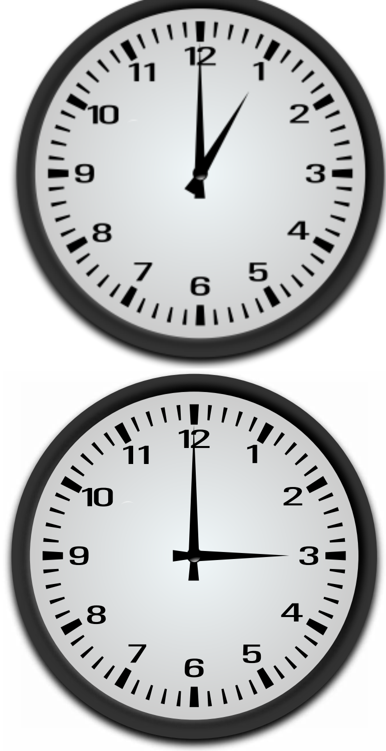


Figure 4.4: Have two hours passed, or 14 hours? Or even a 26 hours? We can't tell. Source: Openclipart.org

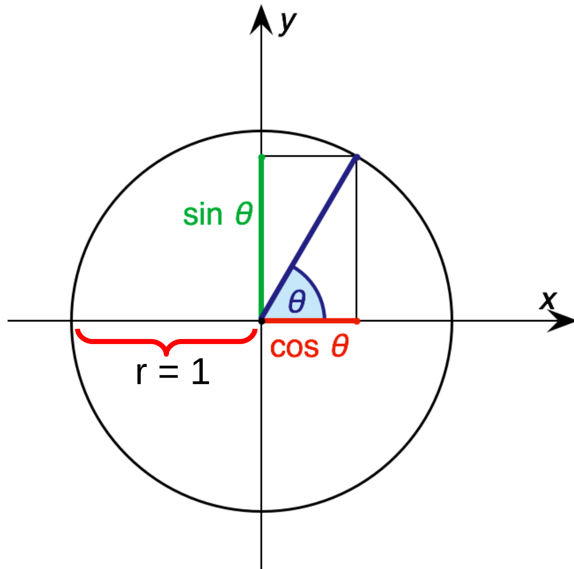
4.5. Trigonometric Functions

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.

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

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.



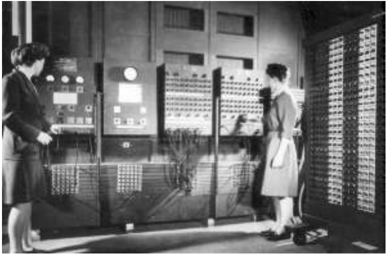


Figure 4.6: Betty Jennings and Frances Bilas operating ENIAC.
Source: Wikimedia Commons

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.

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:

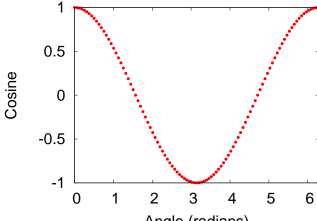
```
./trig > trig.dat
```

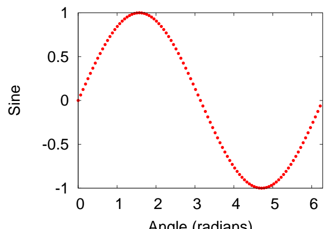
It's hard to see whether your program is doing the right thing by just looking at the numbers. Let's try graphing them. Start up *gnuplot* by typing its name, and then give it this command:

```
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
```





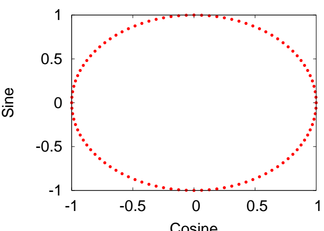


Figure 4.8: Plots of θ versus $cos(\theta)$, θ versus $sin(\theta)$, and $cos(\theta)$ versus $sin(\theta)$.

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.

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.

Before computers and calculators became widely available, the slide rule was widely used for calculations involving logarithms or trigonometric functions.
Source: Wikimedia Commons

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 );
}
```

How many loops are in this roller coaster?
Source: *Wikimedia Commons*

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.

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”);
}
```

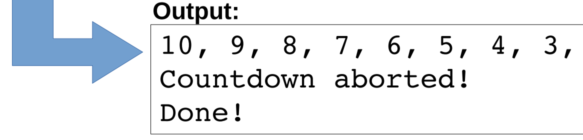


Figure 4.9: Using `break` to stop a loop.

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");
}
```

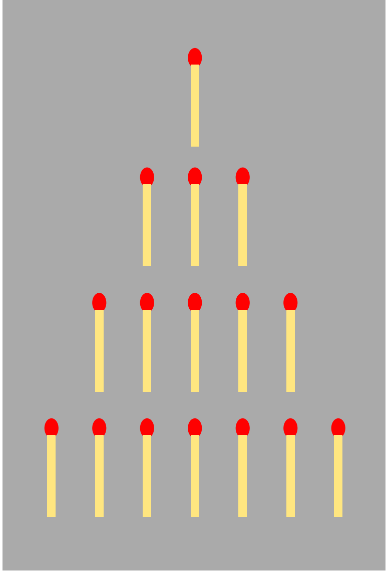


```
10, 9, 8, 7, 4, 3, 2, 1, GO!
```

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".



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

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.

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);
}
```

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 );
```

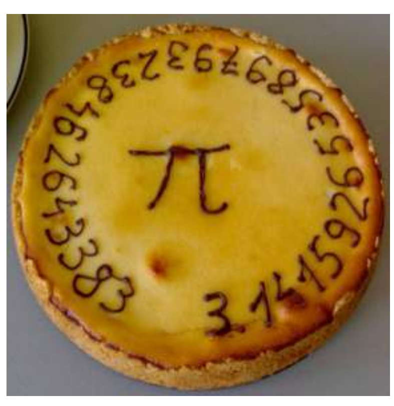
If we ran the example above, it would always print out something, no matter what the value of `i` is. The statements inside a `do-while` loop will always be acted upon at least once. After each trip through the loop, the `do-while` statement's condition is examined to see whether it's satisfied, determining 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?).

[6] See Figure 4.2.

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

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
```

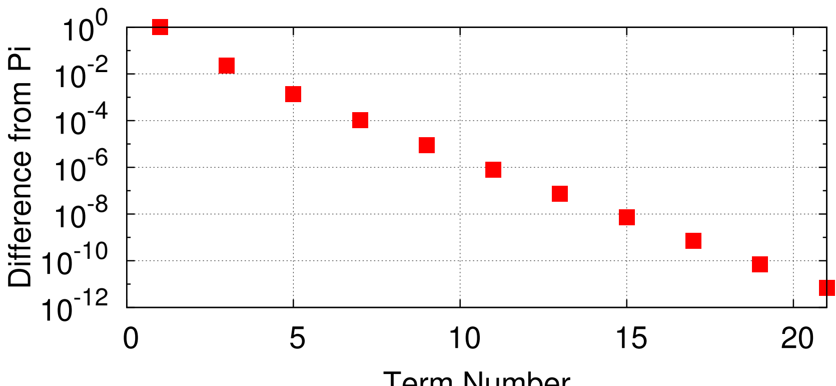


Figure 4.12: The difference betwen our estimate of π and the actual value, as we add more terms to the sum. Note that the vertical scale is logarithmic.

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.

[7] This Numberphile video by Roger Browley shows how log tables were used: https://www.youtube.com/watch?v=VRzH4xBoGdM.

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

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.

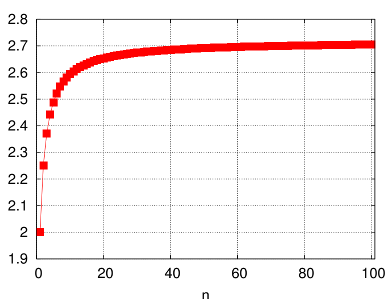


Figure 4.13: This is what a graph of your `interest.cpp` program's output should look like. Notice that the value rises rapidly at first, then levels of to a value approaching e.

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.

[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

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

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.

[9] See Wikipedia for much more information.

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.

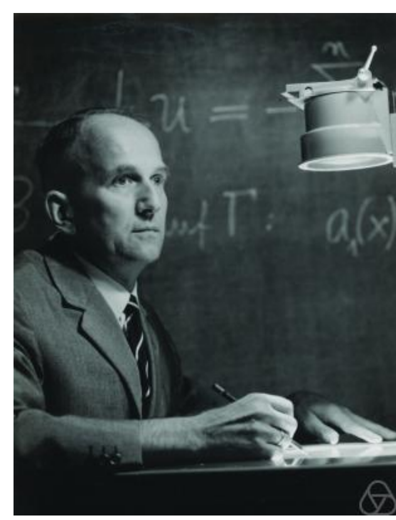
   [10] See https://www.youtube.com/watch?v=5mFpVDpKX70 and https://en.wikipedia.org/wiki/Collatz_conjecture.

   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
```

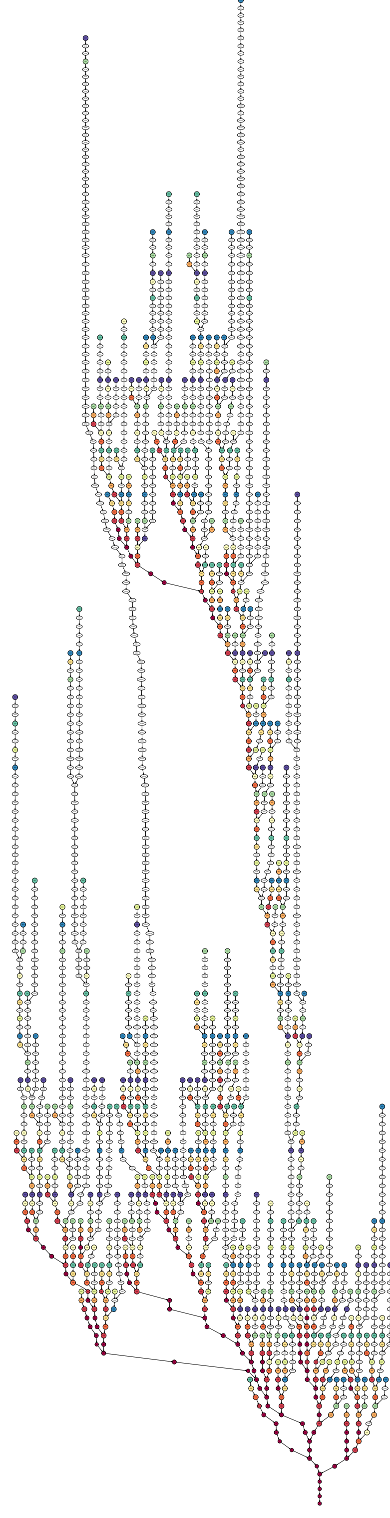


Lothar Collatz (1910-1990)
Source: Wikimedia Commons

   Hint: You can find out whether a number is even by using the

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.



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

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:

[11] This is also sometimes called the "greatest common divisor" or "greatest common denominator".

$$s = \frac{1}{1} + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \cdots$$

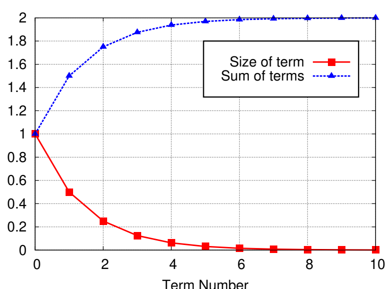


Figure 4.14: In the `sumto2.cpp` program, as we add more terms, each term becomes smaller and their sum converges toward 2.

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:

    - 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

Boy with marbles.
Source: Wikimedia Commons

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.

[12] Be careful! Some versions of C also have a function named `gamma`, but that's something different, and should be avoided.

n	k	$\binom{n}{k}$
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 $\binom{n}{k}$.

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!)

11. The cosine function can be expressed as an infinite sum of terms[13]:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots$$

[13] this is called a Taylor series.

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}.

[14] We need to look at the absolute value because some of the terms are negative.

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`.)

[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)`.

[16] Be careful! Some versions of C also have a function named `gamma`, but that's something different, and should be avoided.

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.

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`.)

[17] Remember to set the sum equal to zero before you start adding to it.

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.)

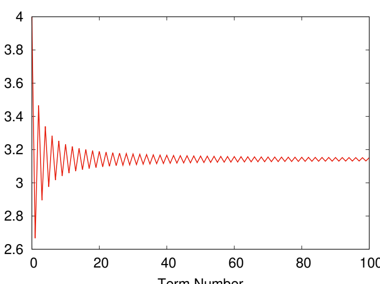


Figure 4.16: A graph of the output of the `leibnizpi.cpp` program, showing how the approximation zooms up and down, eventually settling near the value of π.

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

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.