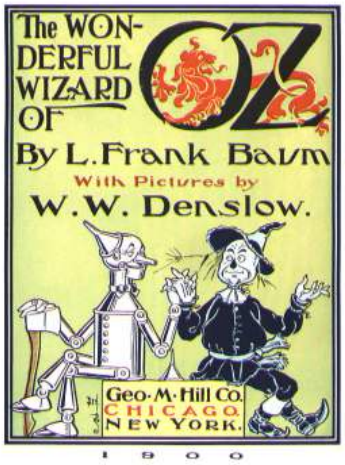
11. Libraries

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.

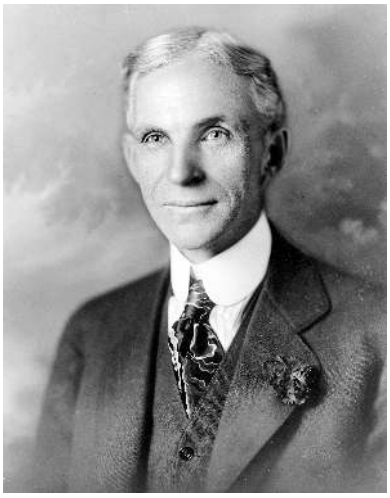
If you write a program that uses a function named `flyingmonkeyspeed`, *g++* first looks at your `cpp` file to see if you've written your own function with that name. If not, *g++* then looks through a standard list of libraries to see if one of them contains a function with that name. If no function 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.

Source: Wikimedia Commons

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:

```
10011100010101110111100010011111001.......
```

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.

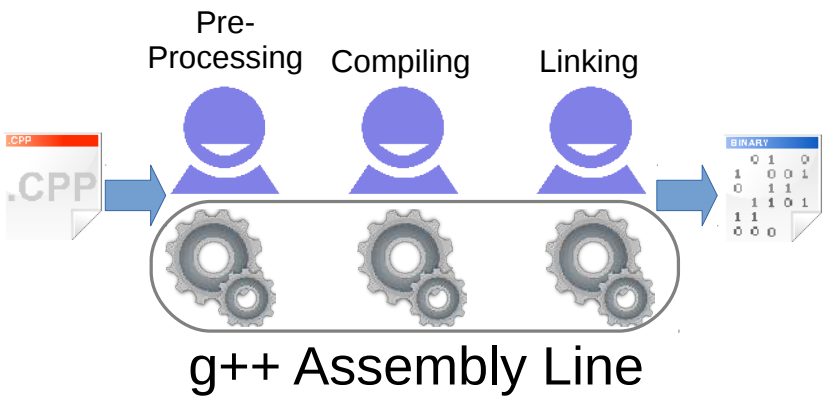


Figure 11.1: g++ does its work in several stages.

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.

[1] Remember that "directory" is just another word for "folder".

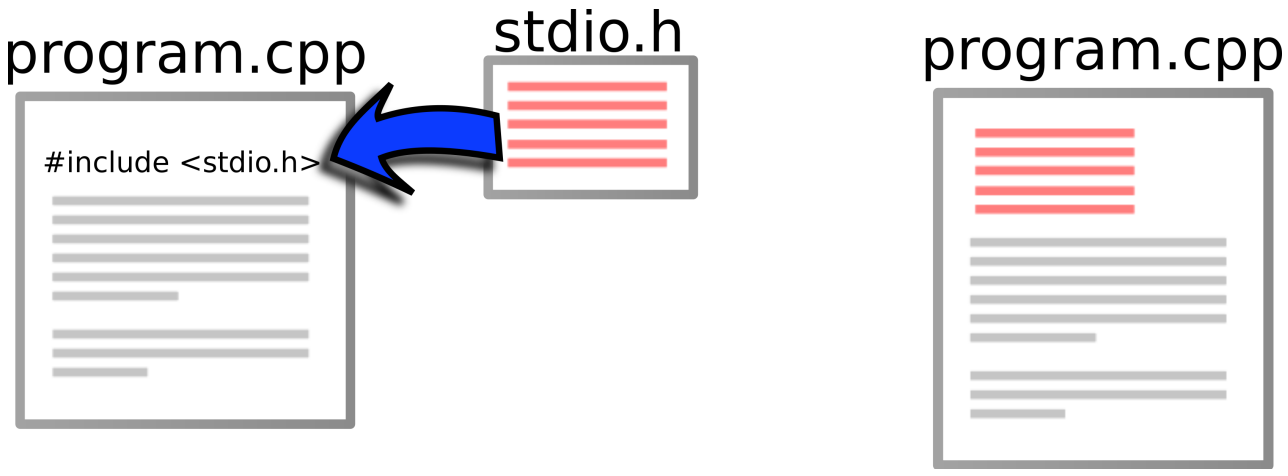


Figure 11.2: The C preprocessor takes the contents of `stdio.h` and inserts them into your program.

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:

- `#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.

[2] Also known as "less than" and "greater than" symbols.

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`.

Source: Wikimedia Commons

11.4. Some Handy Random-Number Functions

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,

"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) );
}
```

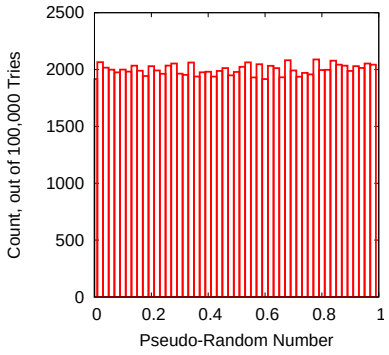


Figure 11.3: Histogram of 100,000 pseudo-random numbers generated by `rand01`.

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.

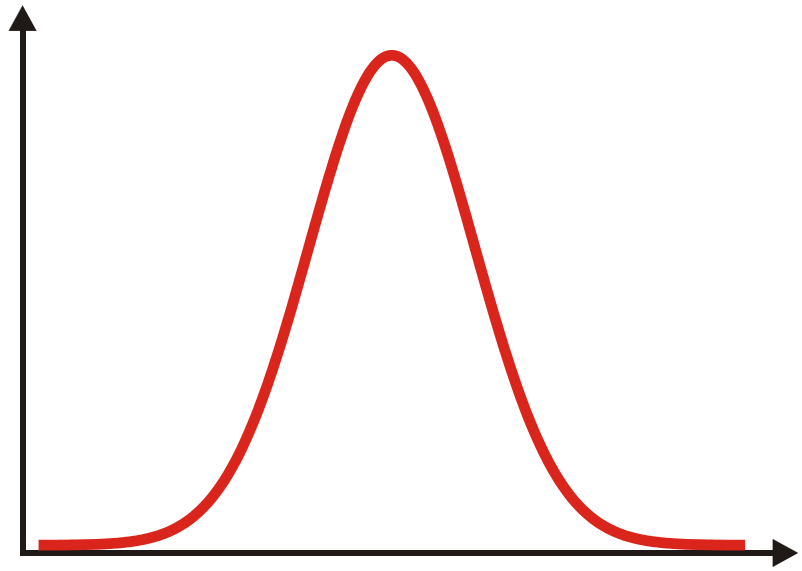


Figure 11.4: A Normal (bell-shaped) distribution. It resembles a slightly-melted witch's hat.

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 );
}
```

[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

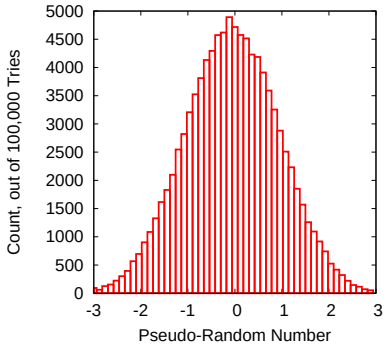


Figure 11.5: Histogram of 100,000 pseudo-random numbers generated by `normal`.

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.

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 );
}
```

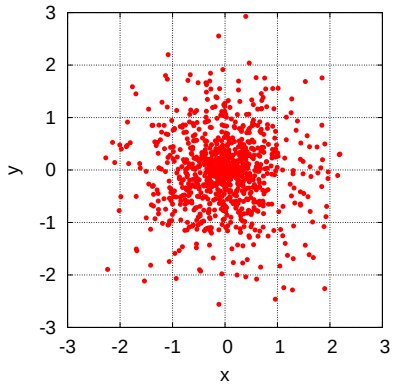


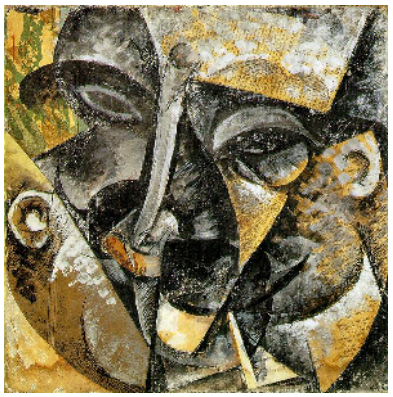
Figure 11.6: The view from the Witch's Munchkin-Scope.

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.



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.

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`.

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:

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 );
}
```

"I've got a header, but I'm still a no-brainer!"

Source: Wikimedia Commons

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.

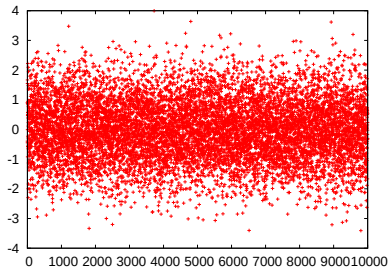


Figure 11.7: Exercise 55 should produce a graph like this.

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.

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

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

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.

Zwei Schlafende Maedchen auf der Ofenbank, by the Swiss artist Albert Anker (1895).
Source: Wikimedia Commons

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.

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.

[4] By this we mean mean *monkey speed*, not *mean monkey* speed, although the latter might be appropriate too.

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.

[5] If you've forgotten how histograms work, take another look at Chapter 7.

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

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 );
}
```

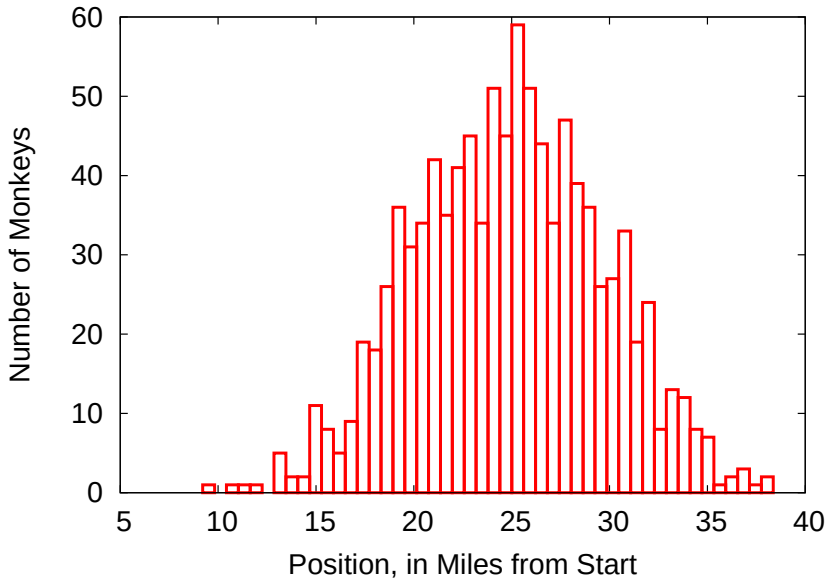


Figure 11.8: The distribution of the monkey swarm after flying for one hour.

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.

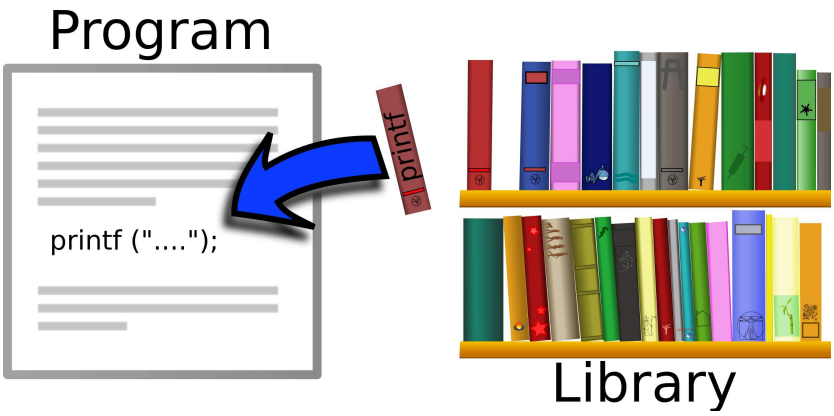
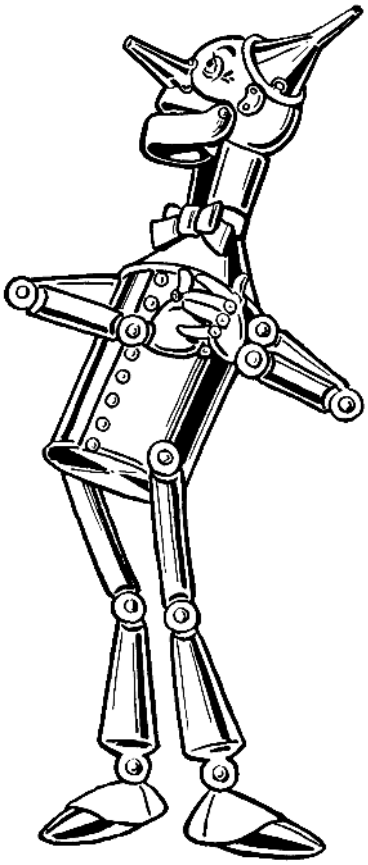


Figure 11.9: *g++* looks through libraries to find any missing functions.

- 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.

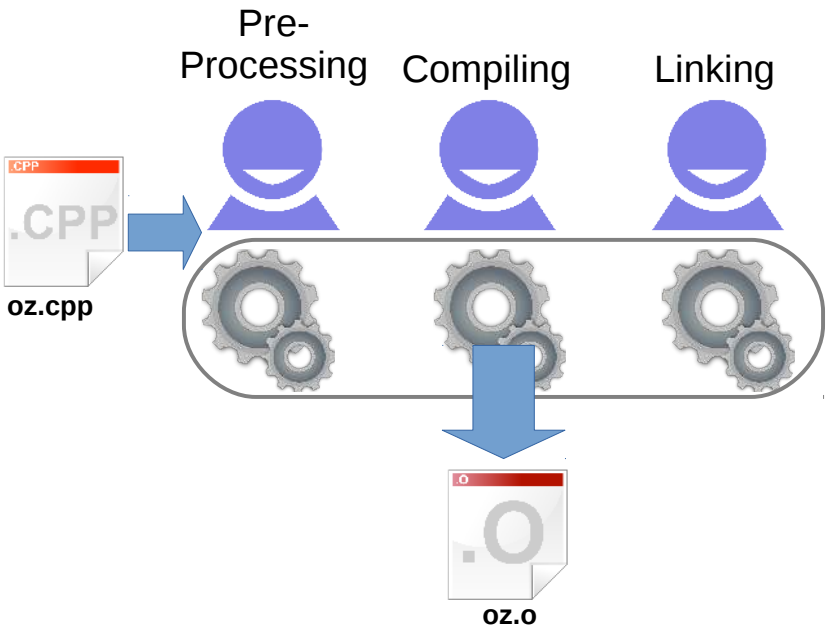


Figure 11.10: An "object" file is created by converting your C code into binary, but not plugging in any functions from libraries.

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:

[6] "`ar`" is short for "archive".

```
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.

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].

[7] In the early days of the GNU project there was a library called "`libiberty.a`", so you could type "`-liberty`".

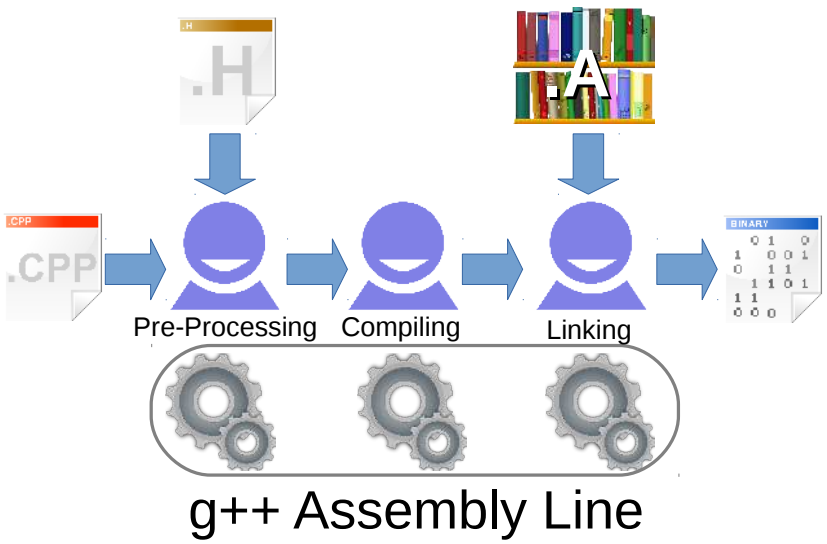


Figure 11.11: The *g++* assembly line processing `monkey.cpp`.

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.

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.

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`.

"Don't be afraid to make your own libraries!"

Source: Wikimedia Commons

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.

[8] *Silver* slippers in the original book.

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

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:

   $$sin(\theta_2) = \frac{n_1}{n_2} sin(\theta_1) \tag{11.1}$$

   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.

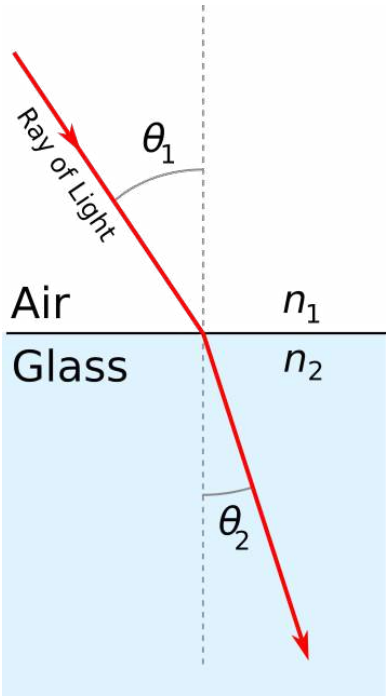
   

   Figure 11.14: Snell's law tells us how light bends when going from one medium to another.
   Adapted from: Wikimedia Commons

   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:

   $$n \simeq 1.5046 + \frac{0.00420}{wavelength^2} \tag{11.2}$$

   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.

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:

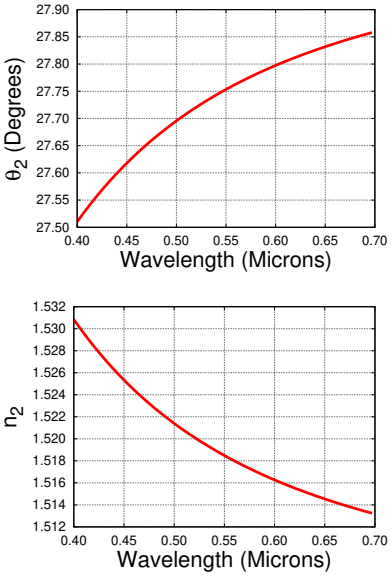


Figure 11.15: Graphs produced by "`plot "optics.dat" with lines`" (top) and "`plot "optics.dat" using 1:3 with lines`" (bottom).

```
#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.

[10] 0.4 to 0.7 microns is the range of wavelengths in visible light.

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!

   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`.

Is your distribution *normal?*
Source: Wikimedia Commons

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.

Look out below!

Source: Wikimedia Commons

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

and speed in Chapter 9: about 3.9 seconds.

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.

```
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.

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.)

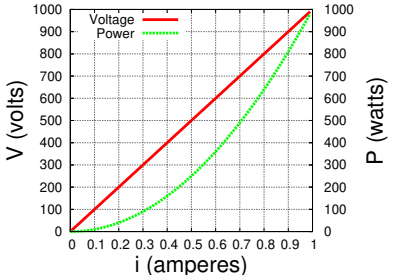


Figure 11.17: Graph of the output from `resistor-check.cpp`.

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.