9. *Functions*

9.1. Introduction

Functions allow you to extend the capabilities of the C compiler.

Despite what you may think after reading the preceding chapters, C is really a very minimal language with only a small vocabulary of about 32 words. This is one reason C has been so successful.

Different types of computer understand different binary instructions, so programs that run on each kind of computer need to be created by a compiler that knows that computer's instruction set. Because making a C compiler is relatively easy (compared to many other computing languages), C is often the first language available when a new type of computer is developed.

Even though the C language is simple, it's powerful because we can extend its abilities by adding "functions" to it. We've already used many of these: `printf`, for example, isn't part of the C language. It's a separate function that has been added. The same is true of the other reading and writing functions we've been using, and the math functions like `sqrt`. All of these are found in standard "libraries" of functions that are usually installed along with the C compiler. The functions in these libraries are themselves written in C. They're essentially pre-compiled snippets of programs, ready to be plugged in where you need them.

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

Figure 9.1: The 32 words of the C language, with an asterisk beside those we've already covered or will cover in this chapter.

Just as you can extend a house by building an extra room, you can build functions that extend the C compiler's capabilities. In this chapter we'll learn how functions work, and see how to create functions of our own.

C's functions let us define simple words to do complicated things. This is especially useful when we have to do a complicated operation over and over again, but it can help us in other ways too. Functions can be re-used in other programs, and using functions can help you avoid programming mistakes.

9.2. What's a Function?

Let's start out by reviewing the kind of functions you've used in math class. Figure 9.2 shows the mathematical function $f(x) = x^2 + 3$. The function is like a machine that takes some raw materials and processes them to produce an output. The function's raw materials are its *arguments*. The function in Figure 9.2 takes one argment, which we've called x here. We could just as easily have written $f(y) = y^2 + 3$. The name we give the argument doesn't matter. It's just a placeholder.

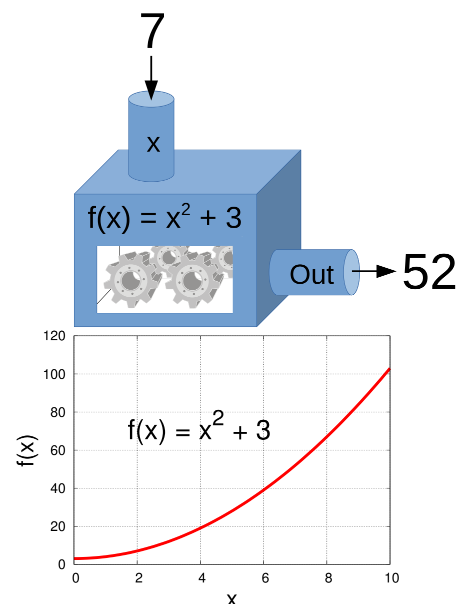


Figure 9.2: You're probably familiar with mathematical functions. A function takes some arguments (inputs), performs some operations on them, then spits out a result.

When we write $f(x) = x^2 + 3$ we're *defining* a function. We've given our function a name, f, we've specifed that it takes one argument (x), and we've said what the function does with that argument to produce an output (square the argument and add three to it). If we put in the value 7, as in Figure 9.2, we'll get out the value 52. We could try a range of different input values and plot the corresponding output values on a graph, as in the lower part of Figure 9.2.

Functions can have more than one argument. Consider the function $g(x, y)$ shown in Figure 9.3. This function takes two arguments (x and y) and produces an output that combines them in a particular way. Functions can have any number of arguments.

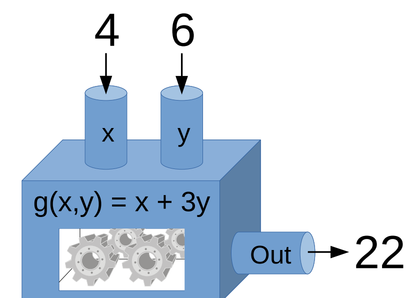


Figure 9.3: A function that takes two arguments, x and y. Given x=4 and y=6 as arguments, the function's output would be 22.

Functions can also make use of *other* functions, as illustrated in Figure 9.4. Here, the function $h(x)$ is defined to be $h(x) = i(x) + 5$, where $i(x)$ is another function, defined as $i(x) = 3x^2$. If we gave $h(x)$ an input of x=2, it would find $i(2)$ = 3×4 = 12, and then add five to this to find that $h(2)$ = 17.

A function in a C program has all the properties we described above:

- A function has a *name*
- The function takes *arguments* and uses them to produce an *output*
- The behavior of a function is described by *defining* the function
- Functions can have *any number* of arguments (in fact we'll see that C function sometimes take no arguments at all!)
- Functions can use *other functions*

As we'll see below, C functions also have some properties that aren't present in mathematical functions.

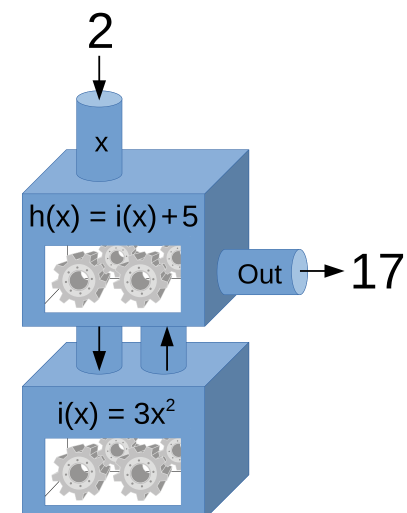


Figure 9.4: The function $h(x)$ shown above uses another function $i(x)$.

Let's look at how we might define a function in a C program. Figure 9.5 shows a C function that takes an input value (an integer we call `x`) and produces an output value that's equal to `x*x + 3`. This is analogous to the mathematical function we saw in Figure 9.2.

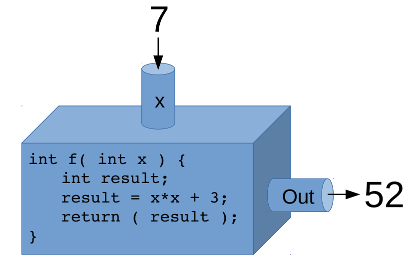


Figure 9.5: A C function named `f` that does the same thing as the mathematical function shown in Figure 9.2.

Program 9.1 shows how we might insert this function at the top of a program, and use it to print some values of the function for various values of its argument. We could plot the program's output with *gnuplot* to create a graph like the one shown in Figure 9.2.

Program 9.1: funcfun.cpp

```
#include <stdio.h>
int f ( int x ) {
  int n;
  n = x*x + 3;
  return ( n );
}

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

Function definition

Using the function

This looks different from anything we've written before. We've added a new section above `int main()`. The new section defines a function named `f`. It says that the function accepts one `int` argument, and returns an `int` value. We then use this new function inside `main()`, in our `printf` statement.

You'll probably notice that the first line of our function definition looks an awful lot like the `int main()` statement that we've been using in all of our programs. That's no coincidence. `main` is a function just like `sqrt`, `printf`, or our new `f` function. It turns out that, in C, almost everything is inside of some function. When we run a C program, the computer looks for a function named "`main`" and does whatever that function tells it to do. We'll see later that we can even give arguments to `main`, as we do with other functions.

Also notice that we've defined our new function *above* `main`. The compiler needs to know about a function before we can use it. One way

to ensure this is to define new functions at the top of the program. We'll see another way to do this later, in Chapter 11, where we'll find out that the line `#include <stdio.h>` tells the compiler about functions like `printf` and `scanf`.

When we use a function in a program, it's as though the program takes a detour into the function and then comes back again with a value:

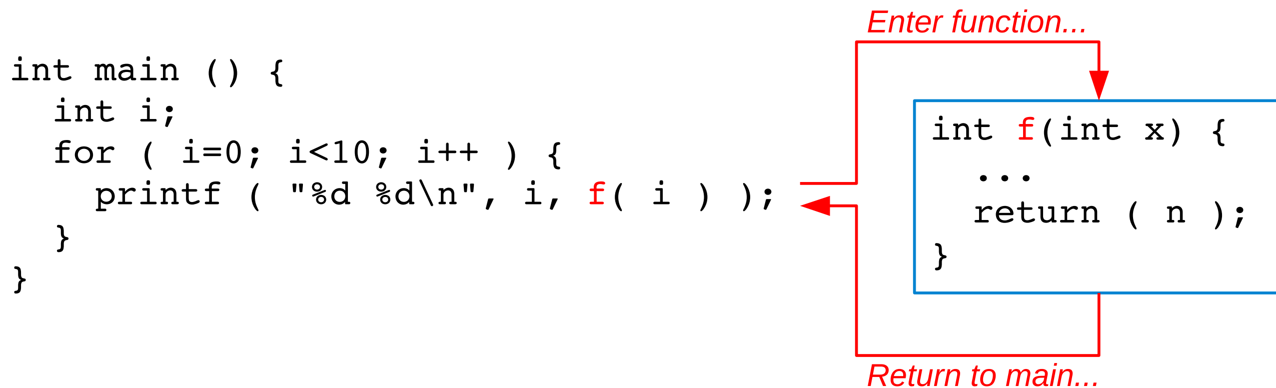


Figure 9.6: The "flow" of the program travels into the function, and then comes back with a result.

Exercise 46: First Function

- Create, compile, and run Program 9.1. Redirect the program's output into a file by running it like this:

  ```
  ./funcfun > funcfun.dat
  ```

- Then use *gnuplot* to plot the program's output, using the *gnuplot* command:

  ```
  plot "funcfun.dat" with lines
  ```

  Does your result look like Figure 9.2?

- Now modify your program so that $f(x) = x^2 + 20$. Compile the program, and run it like this to produce a second data file (`funcfun2.dat`):

  ```
  ./funcfun > funcfun2.dat
  ```

  then use the following *gnuplot* command to plot both files on the same graph:

  ```
  plot "funcfun.dat" with lines, "funcfun2.dat" with lines
  ```

9.3. Function Anatomy

The anatomy of a function definition looks like Figure 9.7. First we need to specify what type of value the function will return. This can be any of the types we use for variables: `double`, `int`, or `char`, for example.

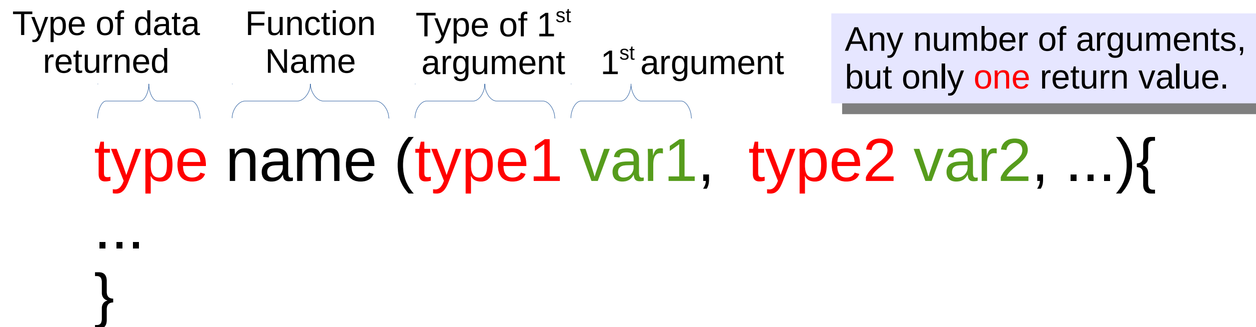
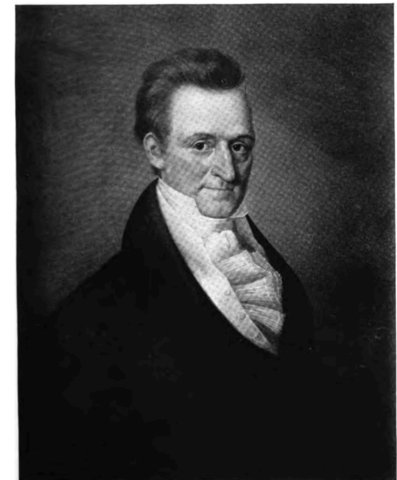


Figure 9.7: The general form of a function definition.

The return value of a C function is like the value you get when you evaluate a function in algebra. The C expression `sqrt(4.0)`, for example, would return the value 2.0. The type of value returned by `sqrt` is a `double`.

By defining the type of value the function will return, you make it possible for the C compiler to check whether you're putting that value into an appropriate variable. If I write a statement like "`x = sqrt(4.0);`" the compiler will check to see if `x` is a `double` variable and give me a warning or an error message if it isn't.

Next we give the new function's name. This must be different from the name of any other function in your program. Function names can contain letters (upper- or lower-case), numbers and underscores. As with variables (see Chapter 2), it's best to start the name of your function with a letter.



Return J. Meigs, Jr., Governor of Ohio, US Postmaster General, and US Senator. As far as I know, C's '`return`" statement wasn't named for him, nor he for it.
Source: *Wikimedia Commons*

After the function's name, we list any arguments and their types. Our $f(x)$ function takes just one argument, and it's an integer. When we use a function in our program, the C compiler checks to make sure we're giving it the right number of arguments, and that the arguments are of the right type. If we've done something wrong, the compiler gives us a warning or an error message.

At the end of our function, as in our $f(x)$ function, we can optionally return a value, but we aren't obligated to return anything. Sometimes a function just does something without returning a value. For example,

we might want a function that just prints some text. If a function doesn't return a value, we specify the function's type as "`void`", like this:

```
void howLong(int hours, int mins, int secs){
  printf("This class is %d seconds long\n",
          hours*3600 + mins*60 + secs);
}
```

Functions that *do* return a value use the `return` statement to do so. In our $f(x)$ example, the statement "`return(n)`" says that the function is done, and sends its result, `n`, back to the `main` function. Functions can only return one value.

Functions don't need to have any arguments, either. The `rand` function is an example of this. When defining a function that takes no arguments, just put an empty pair of parentheses after the function name.

Finally, functions can't be defined inside other functions. We couldn't, for example, define a new function *inside* `main`.

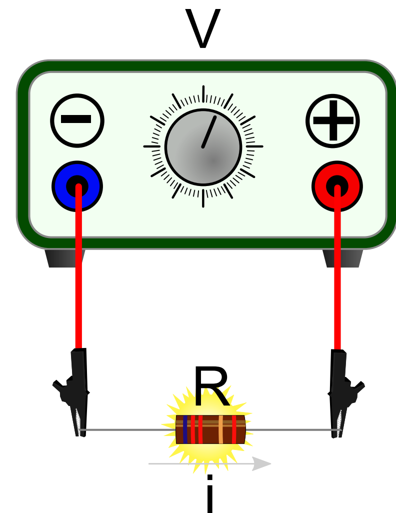


Figure 9.8: An adjustable voltage source is connected to a resistor, causing current to flow through the resistor.

9.4. Functions that Use Other Functions

Consider the apparatus shown in Figure 9.8. Ohm's law tells us that the current (which we represent by the symbol i) flowing through the resistor is given by:

$$i = V/R$$

where V is the voltage across the resistor and R is the resistance. Another law (Joule's Law) tells us that the power output of the resistor (which we represent by p) is given by:

$$p = i^2 R$$

The power is a measure of how fast the resistor is emitting energy, mostly in the form of heat. When we run a current through a resistor, the resistor heats up.

Georg Simon Ohm (left), Alessandro Volta (center), and André-Marie Ampère, for whom the units of resistance, electrical potential, and current are named.
Source: Wikimedia Commons, 1, 2, 3

If we know the voltage and resistance, we can calculate the current, and then we can use the current to calculate the power. If we measure resistance in *ohms*, voltage[1] in *volts*, and current in *amperes*, the power we calculate will be given in units of *watts*.

[1] also called *electrical potential*

Let's write a program that calculates the power output of the resistor at various voltage settings. The result might look like Program 9.2.

Program 9.2: ohm.cpp

```
#include <stdio.h>

double current ( double v, double r ) {
  return ( v/r );
}

double power ( double v, double r ) {
  double i, p;
  i = current ( v, r );
  p = i*i*r;
  return ( p);
}

int main () {
  double r = 100; // ohms.
  double vmin = 0; //volts.
  double vmax = 12; //volts.
  double v, p, vstep;
  int n;

  v = vmin;
  vstep = (vmax - vmin)/100.0;
  for ( n=0; n<100; n++ ) {
    p = power ( v, r );
    printf ( "%lf %lf\n", v, p );
    v += vstep;
  }
}
```

Notice that we've defined two functions, `current` and `power`. The `current` function tells us how much current will flow through the resistor when a given voltage is applied across it. It takes two arguments, `v` and `r`, and returns a value for the current. Because this is a very simple function (it just divides `v` by `r`) we can do the calculation right in the `return` statement. The `current` function just has one line in it!

But what about the `power` function. Shouldn't it have current as one of its arguments, instead of voltage? Sure, we could do it that way, but we want our program to tell us the power for a given voltage, so why not write our `power` function so that it does the calculation for us? Here we've written the `power` function so that it takes voltage and resistance as arguments, then internally uses the `current` function to calculate

the current, before going on to calculate the power and return that.

This makes our `main` program very simple. We just loop through several voltage values and use the `power` function to find the power value at each voltage. The program assumes the resistance is 100 ohms. The program starts at the voltage `vmin` and goes up to the voltage `vmax` in 100 steps. Notice that we calculate the size of each voltage step (`vstep`) before starting the loop, and then add `vstep` to the voltage each time we go around. If we used *gnuplot* to plot the program's output, we'd see a graph like Figure 9.9.

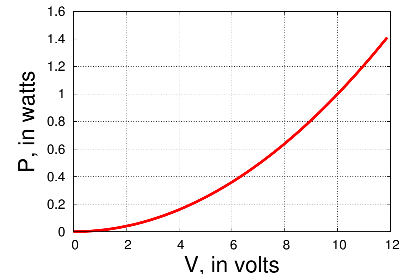


Figure 9.9: Power versus voltage for a 100 ohm resistor.

When you buy a resistor, you need to pay attention to the resistor's *power rating*. Some resistors can only tolerate a power output of $\frac{1}{8}$ watt. Trying to increase the power beyond that would cause the resistor to burn or melt. Resistors that can tolerate more than one watt are often called *power resistors*. Based on our program's output (as graphed in Figure 9.9) we'd need a power resistor that can tolerate at least 1.4 watts if we intend to put 12 volts across it.

Three high-power 100 ohm resistors, with power ratings of 10, 50, and 100 watts. Each has an aluminum case with cooling fins to help dissipate heat.

Source: Wikimedia Commons

Exercise 47: Your Volt Counts!

Create, compile and run Program 9.2. Send the program's output into a file and plot the data using *gnuplot*.

9.5. Variable Scope

If we run two different programs, we don't expect that the variables in one program will interfere with the variables in the other. It would be perfectly OK if one program had an `int` variable named `number` and the other program had a `double` variable with the same name. Variables don't affect things outside the program they're in. A programmer might say that the "scope" of a variable doesn't extend outside the program.

A scope of a different kind: UVa's own Professor Kathryn Thornton replaces solar panels on the Hubble Space Telescope.

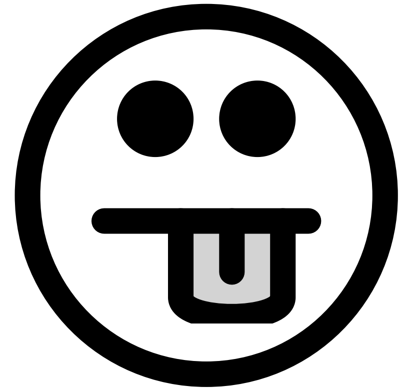
Source: Wikimedia Commons

In fact, in C, the scope of a variable might not even extend to other functions in the *same program*. Each variable in a C program has either a "local" or a "global" scope. All of the variables we've seen so far have local scope. This means that they can only be used inside the function where they're defined. Outside of that function, it's as though these variables don't even exist. (See Figure 9.10 on Page 292.)

The scope of a variable is determined by where it's defined. Variables defined inside a function are local to that function. Take a look at Program 9.3.

Program 9.3: scope.cpp (This won't work)

```
#include <stdio.h>

void printstuff () {
  printf ( "The value of n is %d\n", n );
}

int main () {
  int n = 100;
  printstuff();
}
```



If you tried to compile this program, *g++* would say:

```
scope.cpp: In function 'void printstuff():
scope.cpp:4: error: 'n' was not declared in this scope
```

The variable `n` is only defined inside `main`. As far as the `printstuff` function knows, this variable doesn't even exist.

On the other hand, variables defined outside of any function are *global*. (See Figure 9.10.) They can be used anywhere in your program. Here's a modified version of the program above. All we've done is move one line:

Program 9.4: scope.cpp, with a global variable

```
#include <stdio.h>

int n = 100;

void printstuff () {
  printf ( "The value of n is %d\n", n );
}

int main () {
  printstuff();
}
```

The variable `n` now has a global scope, meaning that every function in your program has access to it. The program will now compile, and do what you expect.

But what happens if you define a global variable, and then define a local variable with the same name? In that case, the local variable takes precedence. Figure 9.10 illustrates this. The function `func2` defines a `double` variable named `height`, even though there's already a global `int` variable with the same name. Inside `func2`, the name `height` will always refer to the local `double` variable, and the global variable of the same name will be inaccessible.

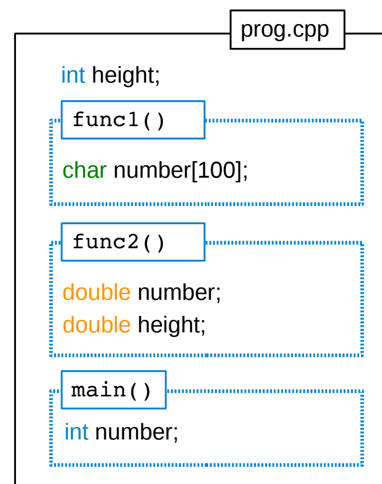


Figure 9.10: The three variables named `number` in `prog.cpp` are completely independent. Each one only exists inside the function where it's defined. Also notice that the `int` variable named `height` has a *global* scope, and can be used by any function. The function `func2`, however, *overrides* the global `height`, replacing it with a local `double` variable that only exists within that function.

It might be tempting to make all of your program's variables global, but avoid this temptation. In general you should use global variables sparingly. If many functions can change a variable's value it's very difficult to keep track of what's going on. It's much better to pass values to functions explicitly, via arguments, rather than to define them globally. For clearer code, it's best to restrict variables to the smallest possible scope. That being said, let's look at how global variables might profitably be used in a program.

9.6. Using Global Variables

Imagine that a rock is dropped from a balloon floating 1,000 meters above the ground, and falls under the influence of earth's gravity. We'll assume that the balloon is close enough to the earth so that we can use a constant value of $g = 9.8m/s^2$ for the rock's acceleration[2]. After some time, t, the rock's speed will be:

[2] We'll also ignore the effects of air resistance.

$$v(t) = gt$$

and it will be at a height h, where:

$$h(t) = 1000 - \frac{1}{2}gt^2$$

Program 9.5 tracks the falling rock for ten seconds. Every hundredth of a second it prints out the rock's current velocity and height. Two functions named `velocity` and `height` calculate those quantities. Notice that both functions are so trivial that they only contain a `return` statement. Both functions need to know the acceleration of gravity, so we store this in a global variable named `g`.

Program 9.5: falling.cpp

```
#include <stdio.h>

double g = 9.8;  // meters per second^2.

double velocity ( double t ) {
  return ( g*t );
}
double height ( double t ) {
  return ( 1000 - 0.5*g*t*t );
}

int main () {
  double t = 0; // elapsed time, in seconds
  int i;

  for ( i=0; i<1000; i++ ) {
    t += 0.01;
    printf ( "%lf %lf %lf\n", t, velocity(t), height(t) );
  }

}
```

Exercise 48: I've Fallen and I Can't Get Up!

Create and compile Program 9.5, then run the program like this:

```
./falling > falling.dat
```

The resulting file should contain three columns representing time, velocity and height. Now plot the height data by starting *gnuplot* and telling it:

```
plot "falling.dat" using 1:3 with lines
```

The phrase `using 1:3` tells *gnuplot* to use the first column (time) for the horizontal values on the graph, and the third column (height) for the vertical values. Does your plot look like Figure 9.11? What does a graph of velocity (instead of height) versus time look like?

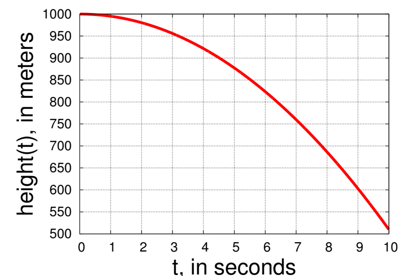


Figure 9.11: The height of a falling stone dropped from 1,000 meters, as a function of time.

9.7. Multiple Returns

Imagine that you're a fighter pilot who's been asked to fly his plane along a very specific (and quite odd) path. After you take off, you're supposed to rise steadily to a height of 1,000 meters, and then fly sinusoidally up and down for a while to evade enemy fire. After that, you're supposed to level off and fly at a constant height.

From the ground, your flight might look like Figure 9.12.

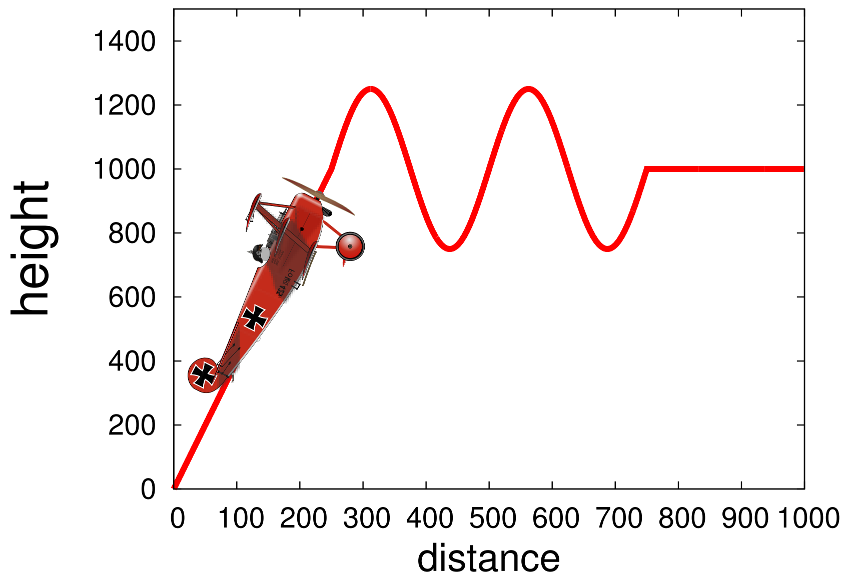


Figure 9.12: Our quirky flight path, which rises linearly for the first 250 meters, undulates for the next 250 meters, and then levels off. Fasten your seatbelts!

Can we write a function that tells us the plane's height as a function of how far the plane has travelled horizontally? One complication is the fact that our flight path has three distinct parts: takeoff, evasion, and cruising. As we see from Figure 9.12 each of the first two parts covers a horizontal distance of 250 meters.

The real "Red Baron", Manfred von Richthofen, the German WWI flying ace who flew a red Fokker triplane.

Source: *Wikimedia Commons*

We might consider writing a function that has an "`if`" statement, like this:

```
if ( x < 250 ) {
        // Takeoff
        ...
} else if ( x >= 250 && x < 750 ) {
        // Evasion
        ...
} else {
        // Cruising
        ...
}
```

Where x is the horizontal distance the plane has travelled.

Using such an "if" statement, we could define a "height" function like this:

Program 9.6: redbaron.cpp

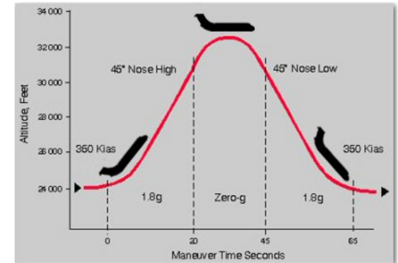
```
#include <stdio.h>
#include <math.h>

double height ( double x ) {

  if ( x < 250 ) {
    return( 1000 * x / 250 );
  } else if ( x >= 250 && x < 750 ) {
    return( 1000 + 250 * sin ( 2 * M_PI * (x-250) / 250 ) );
  } else {
    return( 1000 );
  }

}

int main () {
  int i;
  double x = 0;

  for ( i=0; i<1000; i++ ) {
    x += 1.0;
    printf ( "%lf %lf\n", x, height( x ) );
  }
}
```

Notice that the height function contains more than one return statement. That's OK. The function will use whichever return statement is appropriate, based on the value of x. It's perfectly alright to have a function use different return statements in different circumstances.

It's important to remember that a return statement ends the work done by a function. For example, if we had two lines like these in a function:

```
return ( 1 );
return ( 2 );
```

The function would always return a value of 1, since the first return would tell the function to stop working and return a value.



Aircraft sometimes do fly along odd trajectories. NASA's "Vomit Comet" creates zero-gravity conditions by temporarily flying along a parabolic path. Commercial ventures like "Zero Gravity Corporation" now use the same technique offer the experience to non-astronauts, like Physicist Stephen Hawking.

Source: Wikimedia Commons, Wikimedia Commons

9.8. Circus Physics

Consider the situation depicted in Figure 9.13. A circus performer, "*La Femme Melinite*", is launched from a cannon and flies throught the air, landing in a pool some distance away. As her manager, we'd like to make sure she doesn't miss, so we need to tell the roustabouts where to put the pool for a given cannon angle and initial velocity.

Source: Wikimedia Commons

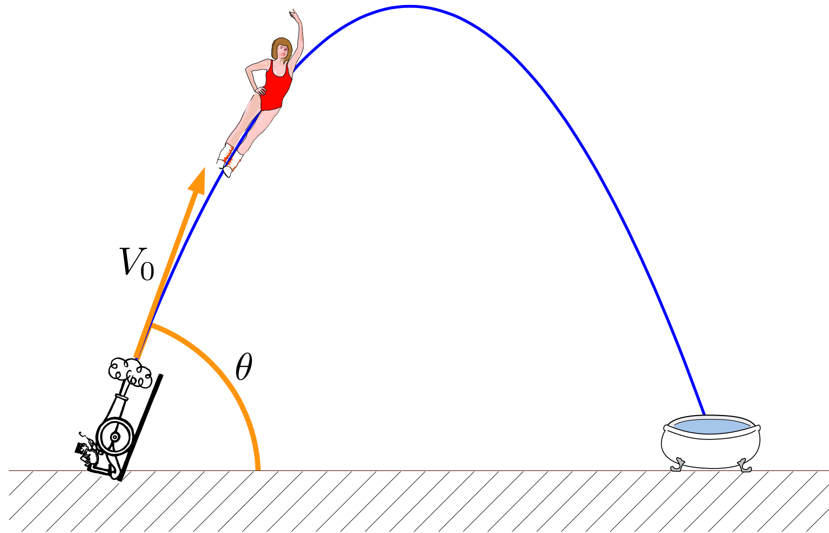


Figure 9.13: A human cannonball, launched at an intial velocity V_0, at an angle θ from the horizontal.

Fortunately, we've had some Physics classes so we know how to find the answer mathematically. The roustabouts, on the other hand, are all English majors. We need to write a computer program that they can use to find out where to put the pool each time they set up the circus.

The program will need to incorporate the mathematical facts of the problem. For example, we know that the total "time of flight" will be given by Equation 9.1, using the y-component of the initial velocity (see Figure 9.14).

$$t_{pool} = \frac{2V_{0y}}{g}, \; g = \text{the acceleration of gravity} \tag{9.1}$$

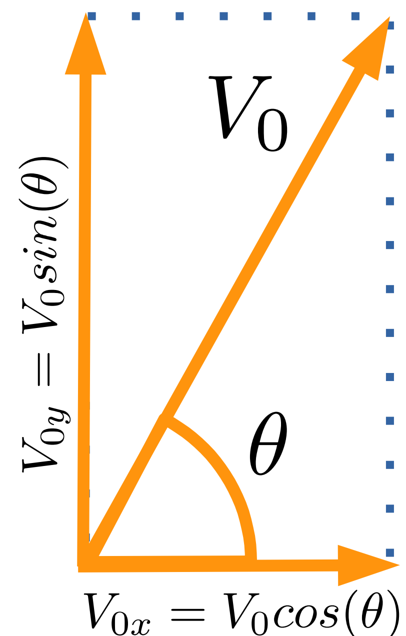


Figure 9.14: Using a little vector math, we can find the x and y components of V_0.

In our program, we could write a function that does the calculation in Equation 9.1. It might look like this:

```
double time_of_flight ( double v0, double angle ) {
  double t;
  t = 2.0 * v0 * sin(angle) / g;
  return ( t );
}
```

Program 9.7 is a little program that uses this function to tell us the time of flight. Notice that we've used a global variable, `g`, to hold the value of the acceleration of gravity. This value will be needed by several of the functions we'll be writing.

The Circus, by Georges Seurat (1891)
Source: Wikimedia Commons

Program 9.7: cannon.cpp

```
#include <stdio.h>
#include <math.h>

double g = 9.81; // Acceleration of gravity.

double time_of_flight ( double v0, double angle ) {
  double t;
  t = 2.0 * v0 * sin(angle) / g;
  return ( t );
}

int main () {
  double vinit;
  double theta;

  printf ( "Enter angle, in radians: " );
  scanf ( "%lf", &theta );
  printf ( "Enter velocity, in m/s: " );
  scanf ( "%lf", &vinit );

  printf ( "Time of flight is %lf sec.\n",
           time_of_flight( vinit, theta ) );
}
```

When we show this program to the roustabouts we're disappointed to find that they don't know how to measure angles in radians. No problem, though. We'll write a function that converts degrees into radians, and let them enter the angle in degrees:

360°
=
2π radians

```
double to_radians ( double degrees ) {
  return ( 2.0 * M_PI * degrees / 360.0 );
}
```

The `to_radians` function just contains one line (a return statement) and doesn't even define any variables. Now, after our program reads the angle, we can convert it into radians by saying "`theta = to_radians(theta)`".

Our Physics education also tells us how to find the maximum height of

La Femme Melinite's trajectory. (We want to make sure she doesn't hit the canvas of the Big Top!)

A circus tent.

Source: Wikimedia Commons

$$t_{peak} = \frac{t_{pool}}{2} \tag{9.2}$$

$$h = V_{0y} t_{peak} - \frac{1}{2} g t_{peak}^2 \tag{9.3}$$

This lets us write a function `max_height` to tell us how high our human cannonball will go.

```
double max_height ( double v0, double angle ) {
  double tpeak;
  double h;
  tpeak = time_of_flight( v0, angle ) / 2.0;
  h = v0*sin(angle)*tpeak - g*tpeak*tpeak/2.0;
  return ( h );
}
```

The last and most important thing we're interested in is the horizontal distance she will travel. That's given by Equation 9.4.

$$d = V_{0x} t_{pool} \tag{9.4}$$

We can express this in C as follows:

```
double range ( double v0, double angle ) {
  double d;
  d = v0 * cos(angle) * time_of_flight( v0, angle );
  return ( d );
}
```

Source: Wikimedia Commons

Putting all of these functions together with our earlier program, we get Program 9.8.

Program 9.8: cannon.cpp, with distance and height

```
#include <stdio.h>
#include <math.h>

double g = 9.81; // Acceleration of gravity.

double to_radians ( double degrees ) {
  return ( 2.0 * M_PI * degrees / 360.0 );
}

double time_of_flight ( double v0, double angle ) {
  double t;
  t = 2.0*v0*sin(angle)/g;
  return ( t );
}

double max_height ( double v0, double angle ) {
  double tpeak;
  double h;
  tpeak = time_of_flight( v0, angle ) / 2.0;
  h = v0*sin(angle)*tpeak - g*tpeak*tpeak/2.0;
  return ( h );
}

double range ( double v0, double angle ) {
  double d;
  d = v0 * cos(angle) * time_of_flight( v0, angle );
  return ( d );
}

int main () {
  double vinit;
  double theta;

  printf ( "Enter angle, in degrees: " );
  scanf ( "%lf", &theta );
  theta = to_radians( theta );

  printf ( "Enter velocity, in m/s: " );
  scanf ( "%lf", &vinit );

  printf ( "Time of flight is %lf sec.\n",
           time_of_flight( vinit, theta ) );

  printf ( "Max height is %lf meters.\n",
           max_height( vinit, theta ) );

  printf ( "Range is %lf meters.\n",
           range( vinit, theta ) );
}
```

Let's try our program out. Imagine that our flying lady is launched at a speed of 60 miles per hour (the world record for a human cannonball was set by someone travelling at about 70 mph). That's approximately equal to 27 meters per second. If the cannon is pointing upward at an angle of 45°, our program tells us the following:

```
Enter angle, in degrees: 45
Enter velocity, in m/s: 27
Time of flight is 3.892331 sec.
Max height is 18.577982 meters.
Range is 74.311927 meters.
```

The Circus, Charles Demuth (1917)
Source: Wikimedia Commons

Note that the Big Top will need to be at least 20 meters tall: as high as a six-story building! Also notice that she'll be in the air for almost four seconds. That's not bad, considering that riders in the Vomit Comet get only 25 seconds of weightlessness during each of the airplane's parabolic leaps (Figure 9.7).

9.9. Passing Values to Functions

The argument names we use when defining a function become local variables inside that function, just like any other local variables that we might define inside it. We can demonstrate that with Program 9.9.

Program 9.9: passing.cpp

```
#include <stdio.h>

void changenum ( int number ) {
  printf ( "Multiplying %d by 1000...\n", number );
  number = number * 1000;
  printf ( "...the result is %d\n", number );
}

int main () {
  int mynum = 1234;
  changenum( mynum );
  printf ( "My number is now %d\n", mynum );
}
```

If we ran this program, we'd see something like this:

```
Multiplying 1234 by 1000...
...the result is 1234000
My number is now 1234
```

What's going on? Why isn't the last number `1234000`?

When we use the function, the computer copies the values of the arguments we give it into the internal, local variables named in the function definition. In Program 9.9, the value of `mynum` (1234) gets copied into the `changenum` function's local variable `number`. Nothing we do to `number` has any affect on the variable `mynum` in `main`.

In fact, the local variables inside functions are, by default, non-existent whenever the function isn't being used. Let's take a look at the computer's memory before, during, and after using the `changenum` function. Figure 9.15 shows what we might see.

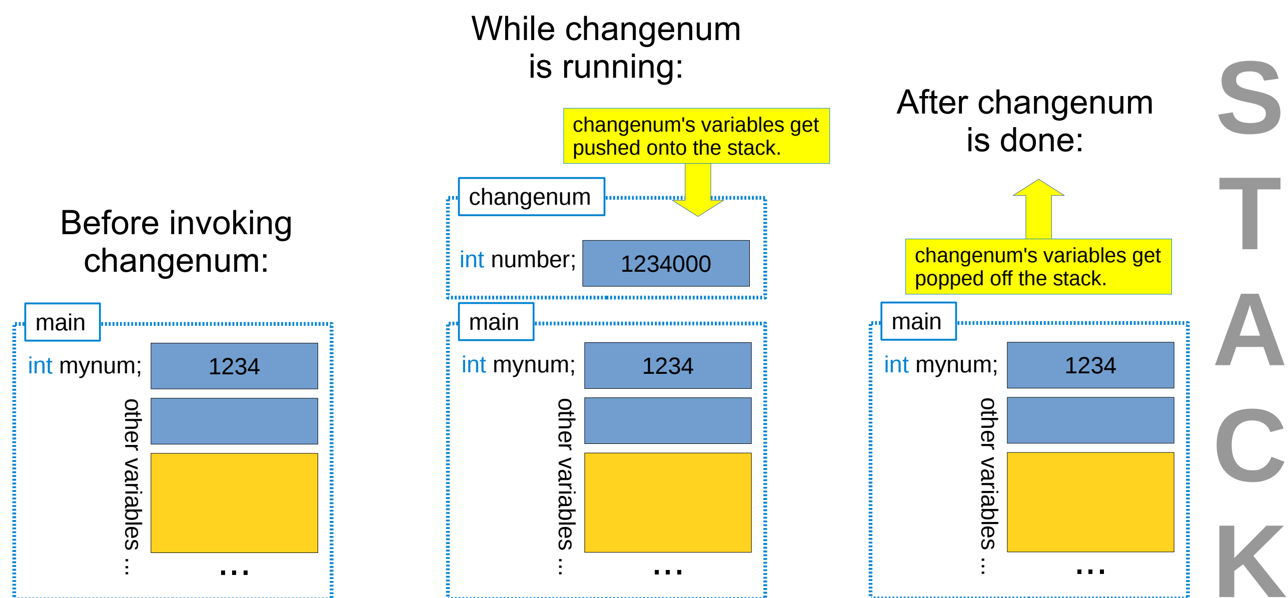


Figure 9.15: The stack before, during, and after using the `changenum` function.

When the function begins, the computer allocates some memory at the top of the stack for each of the function's local variables. When the function finishes, the allocated memory is freed up for other uses (perhaps by the next function that's used). A function's local variables literally disappear when they're not in use.

We can actually see that `mynum` and `number` are stored in different locations by asking our program to print the memory address of each of these variables. Program 9.10 does that by using `&number` and `&mynum` to get the memory addresses, and C's special placeholder for printing memory addresses, "`%p`".

Program 9.10: passing.cpp

```
#include <stdio.h>

void changenum ( int number ) {
  printf ( "number is at %p\n", &number );
  printf ( "Multiplying %d by 1000...\n", number );
  number = number * 1000;
  printf ( "...the result is %d\n", number );
}

int main () {
  int mynum = 1234;
  printf ( "mynum is at %p\n", &mynum );
  changenum( mynum );
  printf ( "My number is now %d\n", mynum );
}
```

If we run Program 9.10 we'll see something like this[3]:

```
mynum is at 0xbfd36ccc
number is at 0xbfd36cb0
Multiplying 1234 by 1000...
...the result is 1234000
My number is now 1234
```

[3] The memory addresses are written as hexadecimal (base-16) numbers.

9.10. Static Variables

As we saw in the preceding section, a function's local variables disappear when the function isn't in use, and are re-created each time we use the function. What if we want to save the value of one of these variables? Maybe, for example, we'd like to have a counter that tells us how many times the function has been called. We could accomplish that with a global variable, but there's also another way to do it.

Source: Wikimedia Commons

Take a look at Program 9.11. It uses the word "`static`" to tell the compiler that we want to retain the value of a variable even when the function isn't being used. Static variables don't live on the stack with other variables. They have their own place in memory, where they don't get wiped out every time the function is called.[4]

[4] The non-static variables we've been using so far are formally called "automatic" variables. If we wanted to, we could explicitly use the word "`auto`" in front of the variable definition to show this, but that's seldom done.

Program 9.11: counter.cpp

```
#include <stdio.h>

void myfunc () {
  static int count = 0;
  if ( count == 0 ) {
    printf ( "This is the first time we've used this function\n" );
  } else {
    printf ( "We've already used this function %d times\n", count );
  }
  count++;
}

int main () {
  int i;
  for ( i=0; i<5; i++ ) {
    myfunc();
  }
}
```

Notice that we can still intialize a `static` variable. In Program 9.11 we set the initial value of `count` to zero. This is only done once, the first time the function is used. The variable won't be reset to zero every time we call the function.

If we ran this program, we'd see something like this:

```
This is the first time we've used this function
We've already used this function 1 times
We've already used this function 2 times
We've already used this function 3 times
We've already used this function 4 times
```

If we had omitted the word "`static`", the variable `count` would be wiped out and reset to zero every time we used the function, so it would just keep repeating "This is the first time we've used this function".

9.11. Passing Addresses

What if we really want one function to be able to change the value of a variable in another function? To do that, we need to know where to find the variable in the computer's memory. As we've seen before, we can

use an & in front of a variable's name to get its memory address. But how do we tell the computer to stick a value into a particular memory address? C provides another symbol, "*" that we can use to help us do this.

Looking for the right (memory) address?
Source: Wikimedia Commons

Program 9.12 is a modified version of Program 9.9. In the new version, instead of giving changenum the *value* of mynum, we give it the *address* of mynum.

Program 9.12: passing.cpp

```
#include <stdio.h>

void changenum ( int *number ) {
  printf ( "Multiplying %d by 1000...\n", *number );
   *number =  *number * 1000;
  printf ( "...the result is %d\n", *number );
}

int main () {
  int mynum = 1234;
  changenum( &mynum );
  printf ( "My number is now %d\n", mynum );
}
```

Memory addresses can be stored in special variables called "pointers". In our new definition of changenum, we say that this function should get a "pointer to an integer" (int *) as its argument. Pointers "point" at the memory location where some data is stored. The * means that this variable is a pointer.

Inside changenum we use the * operator in another way. The expression "*number = ..." means "set the variable at this memory location to ..."[5].

[5] Programmers call & the "referencing operator" and * the "dereferencing operator".

If we ran Program 9.12 we'd see this, showing that we have actually changed the value of mynum:

```
Multiplying 1234 by 1000...
...the result is 1234000
My number is now 1234000
```

9.12. Bouncing Molecules

In 1827, botanist Robert Brown noticed something odd while looking at pollen grains in water, through a microscope. The pollen grains were very small, but he saw that they emitted even smaller particles that we now know were bits containing starch and fat. This in itself was interesting, but Brown was also fascinated by the fact that these tiny particles moved around continously, as though they were alive.

Robert Brown, botanist, by Henry William Pickersgill (1782-1875). For much more information about Brown's work, see this modern-day recreation of it by researchers at Hamilton College.

Source: *Wikimedia Commons*

On further experiments with inorganic matter like bits of glass and granite, he found that those particles displayed the same behavior. What caused them to move? Today we know that the particles Brown observed were being jostled by water molecules, and we call this phenomenon "Brownian Motion".

Program 9.13 simulates the motion of a tiny particle floating on the surface of some water. It begins by picking a random starting position for the particle by setting the x and y coordinates of the particle's position to random numbers between zero and one.

The program then tracks the particle through 10,000 collisions. Each collision moves the particle by some random amount. The function `move` takes the particle's current x and y coordinates and changes them to new values by adding a random amount between -0.5 and 0.5. Notice that we give `move` the *addresses* of `x` and `y`, making it possible for the function to change the values of these variables, as described in Section 9.11 above.

The program prints each new position, so we could plot the particle's path if we wanted to. Figure 9.16 shows the path of a typical particle.

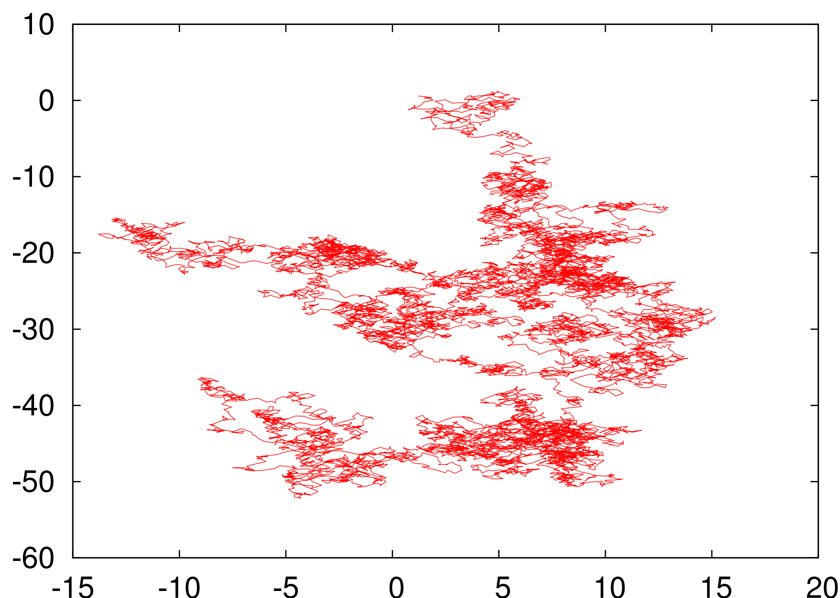


Figure 9.16: The path of a typical particle in our brownian motion simulation. For a deeper investigation of the mathematics of random walks, see this video from the PBS show "Infinite Series": https://www.youtube.com/watch?v=stgYW6M5o4k

Program 9.13: brownian.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void move ( double *x, double*y ) {
  *x = *x + rand()/(1.0+RAND_MAX) - 0.5;
  *y = *y + rand()/(1.0+RAND_MAX) - 0.5;
}

int main () {
  double x, y;
  int i;

  srand(time(NULL));

  // Pick a random initial position:
  x = rand()/(1.0+RAND_MAX);
  y = rand()/(1.0+RAND_MAX);

  // Move around:
  for ( i=0; i<10000; i++ ) {
    move( &x, &y );
    printf ( "%lf %lf\n", x, y );
  }
}
```

9.13. Passing Arrays to Functions

Source: Wikimedia Commons

Sometimes we want a function to operate on an array. We can do that, as shown in Program 9.14, which finds the biggest element of an array of doubles. We could use this to find the most heavily-laden coal car in our coal train example from Chapter 6, for example.

Program 9.14 fills an array with random values, and then uses the function maxelement to find the element that contains the largest value.

Program 9.14: findmax.cpp

```
#include <stdio.h>
#include <stdlib.h>

int maxelement ( int size, double array[] ) {
  double max;
  int imax;
  int i;
  for ( i=0; i<size; i++ ) {
    if ( i == 0 ) {
      max = array[i];  // Use first number as first guess.
    } else {
      if ( array[i] > max ) {
        max = array[i];
        imax = i;
      }
    }
  }
  return ( imax );
}

int main () {
  double array[100];
  int i;
  for ( i=0; i<100; i++) {
    array[i] = rand();
    printf ( "%d %lf\n", i, array[i] );
  }

  printf ( "The biggest element is number %d\n",
           maxelement( 100, array ) );
}
```

Notice that we tell the compiler that one of maxelement's arguments will be an array by putting [] after the variable name. Also notice that we need to tell the function how big the array is. The size argument to maxelement tells the function how many elements are in the array.

Exercise 49: Dot Products

Imagine you have two 3-dimensional vectors, **A** and **B**. Each of these can be represented as 3-element array in C. In mathematics, the "dot product" of two 3-d vectors is

$$\mathbf{A} \cdot \mathbf{B} = \sum_i A_i B_i \quad (9.5)$$

or, writing out the sum:

$$\mathbf{A} \cdot \mathbf{B} = A_0 B_0 + A_1 B_1 + A_2 B_2 \quad (9.6)$$

Write a function that takes two 3-element `double` arrays as arguments and returns their dot product as a `double` value.

Test your function with a program that multiplies these two arrays together and prints out their dot product:

```
double a[3] = { 1.0, 2.0, 3.0 };
double b[3] = { 4.0, 5.0, 6.0 };
```

9.14. The Chaos Game

Let's write another program that passes arrays to a function. This program will play "The Chaos Game". the rules of this game are:

1. Pick three reference points on a piece of paper and label them ***P*1**, ***P*2**, and ***P*3**.
2. Draw another point (let's call it ***b***) anywhere on the paper.
3. Randomly choose one of the reference points. For example, you could roll a 6-sided die and pick ***P*1** if you get **1** or **2**, ***P*2** if you get **3** or **4**, and ***P*3** if you get **5** or **6**.
4. Draw a new point halfway between ***b*** and the reference point you picked. This point becomes the new ***b***.
5. Go back to step 3 and repeat.

In Discordianism, Eris is regarded as the goddess of chaos. She says "I am the substance from which your artists and scientists build rhythms. I am the spirit with which your children and clowns laugh in happy anarchy. I am chaos. I am alive, and I tell you that you are free." (from the *Principia Discordia*).
Source: Wikimedia Commons

Doing this by hand would get boring pretty quickly, so let's write a computer program to do it for us. Program 9.15 repeats steps 3 and 4 10,000 times, printing the new coordinates of ***b*** each time.

At the top of the program we define our three reference points. Each point is represented by a 2-element array containing the point's x and y coordinates. The function `movehalf` takes the point b (also represented by a 2-element array) and moves it half of the way toward one of our reference points.

Notice that we give b to `movehalf` as an array. You might remember that, in Chapter 8, we learned that C programs interpret an array's name (without an element number after it) as the *memory address* of the array. This means that when we give a function an array as one of its arguments, the function is able to change the values of the array elements, just as when we give a function the memory address of a single variable (which we saw above, in Section 9.11).

The `main` function picks random starting values for the coordinates of our point b, then uses the rules of the Chaos Game to move this point around. Instead of rolling a die, the program generates a random number between zero and one. If this number is less than **1/3** the program moves the point halfway to point $P1$. If it's between **1/3** and **2/3** it moves toward $P2$. If it's between **2/3** and **1**, it goes toward $P3$.

We could save the program's output in a file by typing "`./chaos > chaos.dat`", and then we could graph these points with the *gnuplot* command "`plot "chaos.dat" with dots`". The result is shown in Figure 9.17.

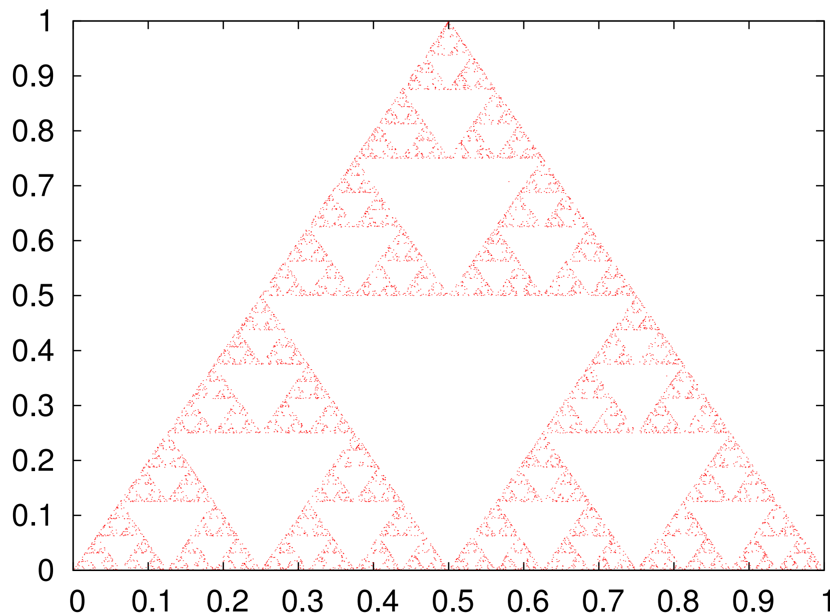


Figure 9.17: A "Sierpinski Triangle" (or "Sierpinski Gasket") drawn by playing The Chaos Game. Telling *gnuplot* to use "dots" causes it to draw small points instead of symbols. For more on The Chaos Game see this Numberphile video: https://www.youtube.com/watch?v=kbKtFN71Lfs

You might well be surpised by this result! The shape you see is called a Sierpinski Triangle (or Sierpinski Gasket). Since we picked a random direction each time, you might have expected the points to be spread evenly around the page. In fact, no matter where you start on the page, the points will eventually be "attracted" to the red areas on the graph. This shape is an example of a "chaotic attractor". Even though we can't predict where a given point will land on the graph, the overall pattern of all the points is very orderly and well-defined. This is an example of order emerging spontaneously from randomness. Such phenomena are common in the natural world, where simple underlying rules can lead to intricately beautiful structures.

Program 9.15: chaos.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

double p1[2] = {0,0};
double p2[2] = {0.5,1.0};
double p3[2] = {1.0,0.0};

void movehalf ( double b[], double point[] ) {
  b[0] = b[0] + 0.5*( point[0] - b[0] );
  b[1] = b[1] + 0.5*( point[1] - b[1] );
}

int main () {
  double r, b[2];
  int i;

  srand(time(NULL));

  b[0] = rand()/(1.0+RAND_MAX);
  b[1] = rand()/(1.0+RAND_MAX);

  for ( i=0; i<10000; i++ ) {

    r = rand()/(1.0+RAND_MAX);
    if ( r < 1.0/3 ) {
      movehalf( b, p1 );
    } else if ( r < 2.0/3 ) {
      movehalf( b, p2 );
    } else {
      movehalf( b, p3 );
    }

    printf ( "%lf %lf\n", b[0], b[1] );
  }
}
```

9.15. Command-Line Arguments

If `main` is just a function, can we give it arguments? Yes we can, but they must always be a particular pair of arguments. It turns out that the arguments given to `main` contain anything you type on the command line after the name of your program. These extra things are called "command-line arguments".

Take a look at Program 9.16. This program uses several new concepts. First, notice that `main` now has two arguments, `int argc` and `char *argv[]`. The `main` function must always have either these two arguments or none at all. The first argument, `argc`, tells the program how many arguments you typed on the command line when you ran the program. The second argument is an array of character strings, each element of which contains one of the command-line arguments.

Program 9.16: args.cpp

```
#include <stdio.h>
int main ( int argc, char *argv[] ) {
  int i;

  for ( i=0; i<argc; i++ ) {
    printf ( "argv[%d] = \"%s\"\n", i, argv[i] );
  }
}
```

Let's see what happens when we run the program. If we just type `./args` the program says:

```
argv[0] = "./args"
```

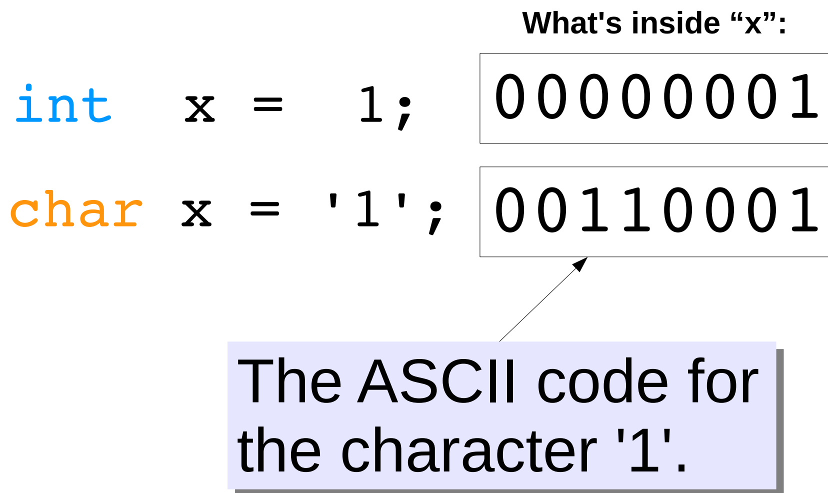
`argv[0]` will always contain the name of the program itself, as it's typed on the command line. Now look what happens if we type `./args hello 1 2 3` on the command line:

```
argv[0] = "./args"
argv[1] = "hello"
argv[2] = "1"
argv[3] = "2"
argv[4] = "3"
```

We could use these command-line arguments to control our program's behavior if we wanted to.

Notice, however, that all of the elements of `argv` are character strings,

not numbers. In the example above, `argv[1]` is equal to the *character string* `"1"`, not the number 1. We can see how these differ by looking at how each value is stored in the computer's memory:



C's standard libraries provide a pair of functions for converting strings to numbers: `atof` and `atoi`. The `atof` function converts a character string into a floating-point number (a `double`), and `atoi` converts a string into an `int`. In order to use these functions, we need to add `#include <stdlib.h>` at the top of the program. In the next section we'll look at a program that uses these functions.

Here's a simple program that illustrates the use of `atoi` to convert a command-line argument into an `int`. The program counts up to a number given on the command line. For example, if you said:

```
./countto 10
```

the program would count to ten.

Program 9.17: countto.cpp

```
#include <stdio.h>
#include <stdlib.h>
int main ( int argc, char *argv[] ) {
  int i,n;
  n = atoi( argv[1] );
  for ( i=0; i<=n; i++ ) {
    printf ( "%d\n", i );
  }
}
```

Convert argument to `int`

9.16. Command-Line Cannon

Let's use command-line arguments to write an improved version of our earlier "cannon" program. Program 9.18 shows a modified version of Program 9.8, omitting the definitions for functions other than `main`. (The other functions will be the same for both programs.)

The new version of the program lets us enter the angle and intial velocity on the command line when we run the program, instead of asking the user for these values. For example, we could type:

```
./cannon 45 27
```

to point the cannon at a 45°angle and specify an initial velocity of 27 m/s.

Program 9.18 first checks to see if you've given it the right number of command-line arguments by looking at the value of `argc`. We want to make sure the user has given values for `theta` and `vinit`. If not, the program prints out a friendly usage message and stops the program.

We can stop the program at any time by using the "`exit`" function, which is part of C's standard library of functions. `exit` takes one argument: an integer number specifying the exit status of the program. This can be any number you like, but usually anything other than zero means that the program failed. You can put an "`exit(0);`" statement at the end of your programs, but it's not necessary.

Notice that we check to make sure `argc` is equal to 3. Why 3? Won't there be only two arguments, `theta` and `vinit`? The `argv` array actually contains one extra thing: the name of the program itself. If we type "`./cannon 45 27`", the elements of `argv` look like this:

```
argv[0] = "./cannon";
argv[1] = "45";
argv[2] = "27";
```

Program 9.18 uses `atof` to convert the command-line values of `theta` and `vinit` into numbers.

Finally, program 9.18 uses the program name as part of the friendly error message it prints if the user doesn't supply enough command-line arguments.

Program 9.18: cannon.cpp, with command-line arguments

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

// Other functions go here....

int main ( int argc, char *argv[] ) {
  double vinit;
  double theta;

  if ( argc != 3 ) {
    printf ( "Syntax: %s theta vinit\n", argv[0] );
    exit(1);
  }

  theta = atof( argv[1] );
  theta = to_radians( theta );

  vinit = atof( argv[2] );

  printf ( "Time of flight is %lf sec.\n",
           time_of_flight( vinit, theta ) );

  printf ( "Max height is %lf meters.\n",
           max_height( vinit, theta ) );

  printf ( "Range is %lf meters.\n",
           range( vinit, theta ) );
}
```

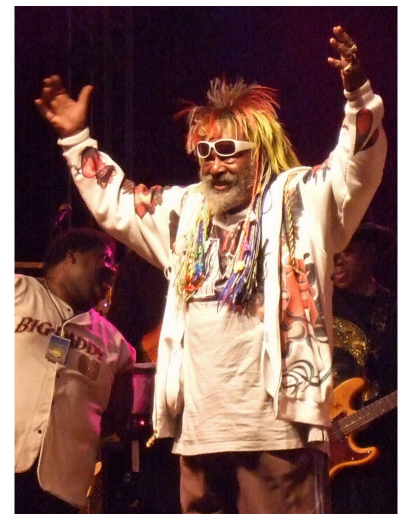
Exercise 50: Hang Time

Write a program like Program 9.18 using the `time_of_flight` function from Program 9.8. The program should accept two command-line arguments, `theta` and `vinit`, and it should print the time of flight based on the values supplied by the user.

Keeping the angle at 45°, run your program repeatedly to find the minimum initial velocity (to the nearest m/s) the acrobat would need if she wanted to remain in the air for at least 25 seconds (matching a ride on the Vomit Comet).

9.17. Passing Functions to Other Functions

We've written a lot of programs that print a list of x and y values. The "Red Baron" program (Program 9.6) earlier in this chapter is a recent example. These programs loop through a bunch of x values, compute the y value for each, and print the results. The value of y is given by some function of x. The function might be something simple like `sqrt(x)` or it might be something complicated, like the Red Baron's flight path.



George Clinton, "the Godfather of Funk".
Source: Wikimedia Commons

No matter what the function is, though, we often do the same thing with it: calculate its value for several x values and print the result. Wouldn't it be nice if we had a function that would accept the name of a "y-generating" function, and print a list of x and y values using it? Let's make one!

Take a look at Program 9.19. As you can see, its `main` just contains one statement. This statement uses the function `plotit` to produce a list of 100 x and y values for $y = \sqrt{x}$, with values of x ranging between zero and 500. The first argument to `plotit` is the *address* of the `sqrt` function. Just as we passed the addresses of variables to a function in Section 9.11, we can also pass the address of a function.

At the top of the program is the `plotit` function. To tell `plotit` to expect a function address, we write one of its arguments as:

```
double (*func)(double)
```

which means "this argument will be the address of a function that takes a `double` as its only argument and returns a `double`". This kind of

argment is called a "function pointer" because it points to the memory address of a function. In general, such an argument will have this form:

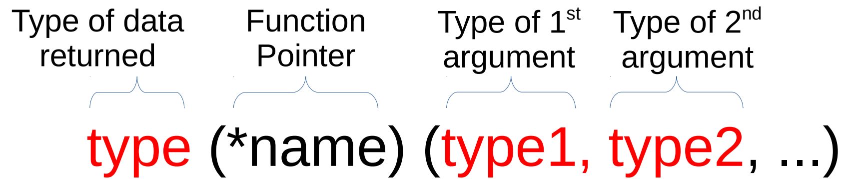


Figure 9.18: General form of a function pointer.

Inside `plotit`, we can use the name `func` to refer to the function, no matter what it really is.

When we use the `plotit` function, we give it the address of the function to be plotted by writing the function's name, just as we do for arrays.

We could replace `sqrt` with `cos` to get a list of values for $y = cos(x)$, or we could use `exp` to get $y = e^x$. We can use any function that takes a `double` as its only argument and returns a double.

Program 9.19: funcplot.cpp

```
#include <stdio.h>
#include <math.h>

void plotit ( double (*func)(double), int nsteps, double xmin, double xmax ) {
  int i;
  double x, step;
  step = (xmax - xmin)/nsteps;

  x = xmin;
  for ( i=0; i<nsteps; i++ ) {
    printf ( "%.10e %.10e\n", x, func(x) );
    x += step;
  }
}

int main () {
  plotit( sqrt, 100, 0, 500 );
}
```

Finally, notice that `plotit` uses the format `%.10e` when printing numbers. As we've seen before, the `.10` tells `printf` to print ten decimal places. The `plotit` function uses `e` instead of `lf` to tell `printf` to print the numbers in scientific notation. This gives our function the ability to print a wide range of numbers.

9.18. Using qsort for Sorting

Licorice Allsorts are among the author's favorite candies. Yum!
Source: *Wikimedia Commons*

In Chapter 6 we looked at the "Bubble Sort" algorithm, which we used for sorting the elements of an array. Now let's look at a faster, more flexible way of sorting array elements.

The C standard libraries contain a function named `qsort` (for "Quick Sort") that can be used to sort any kind of array. To use it, we give `qsort` the name of the array to be sorted, and the address of a function (written by us) for comparing any two array elements. When the function compares two elements (let's call them element a and element b) it should return zero if the two elements are the same, -1 if a is less than b, or 1 if a is greater than b.

Since `qsort` can be used to sort any type of array, we can't assume that the array elements will be `int`s or `double`s or any other specific type. Because of this, `qsort` passes a and b to our comparison function as `void` variables (variables without any specific type), and it's up to the comparison function to figure out how to use them.

Here's what a comparison function for comparing two integers might look like:

```
int compare_int(const void *i1, const void *i2){
  int a, b;

  a = *(int *)i1;
  b = *(int *)i2;

  if ( a<b ) {
    return (-1);
  } else if ( a>b ) {
    return (1);
  } else {
    return (0);
  }
}
```

As you can see, the two arguments given to the comparison function are the addresses (notice the asterisks) of two `void` variables (meaning variables of any type). The function first needs to convert those into integers. It does this by converting the `void` addresses into `int` addresses, then it puts another asterisk on the left to get the actual integer values stored at those addresses and store them in the variables `a` and `b`. Then it's just a straightforward "`if`" statement to compare the two

numbers and return zero, -1, or 1, whichever is appropriate.

Program 9.20 reads an unsorted list of integers from a file and uses the `qsort` function to sort them. The unsorted numbers are in a file named `unsorted.dat`. The program will read as many numbers as are in the file, up to a maximum of 1,000 numbers. After the numbers are sorted, the program prints them in their sorted order, from smallest to largest.

Program 9.20: sortit.cpp

```
#include <stdio.h>
#include <stdlib.h>
int compare_int(const void *i1, const void *i2){
  int a, b;

  a = *(int *)i1;
  b = *(int *)i2;

  if ( a<b ) {
    return (-1);
  } else if ( a>b ) {
    return (1);
  } else {
    return (0);
  }
}

int main ( int argc, char *argv[] ) {
  FILE *input;
  const int nmax = 1000;
  int i, n=0, numbers[nmax];

  input = fopen( "unsorted.dat", "r" );
  while ( n<nmax &&
          fscanf( input, "%d", &numbers[n] ) != EOF ) {
    n++;
  }
  fclose ( input );

  qsort( (void *)numbers, n, sizeof(int), compare_int );

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

Program 9.20 gives the `qsort` function four arguments:

Coat of arms of "Sort", a town in Catalonia, Northwest Spain. Its name means "luck" in the Catalan language.
Source: Wikimedia Commons

1. The name of the array to be sorted, cast as a "`(void *)`". This gives `qsort` the memory address of the array, without specifying any particular variable type.
2. The number of elements to be sorted. Note that this doesn't have to be all of the elements in the array. In the example above, if `unsorted.dat` only contained 100 numbers, then `n` would be 100, even though the array has 1,000 elements.
3. The size (in bytes) of each element of the array. Since `qsort` doesn't know what kind of elements it's sorting, we need to tell it how big they are. Here we use the `sizeof` statement that we introduced in Chapter 6.
4. Finally, we give `qsort` the name of our comparison function. In this case, it's just the `compare_int` function we wrote above.

When our comparison function compares two elements, we're free to define what we mean by "greater than", "less than", or "equal". Why would we need this flexibility? Imagine, for example, that we had a coal train with many cars, each with a different amount of coal, and each destined for a different customer. We might have an array of customer IDs. We could just sort the array in order of increasing ID number, but we might sometimes want to sort the list of IDs based on how much coal they ordered, or by how many miles it is from the coal mine to the customer. The `qsort` function gives us the flexiblity to do that[6].

[6] Later on, in Chapter 12, we'll see that C lets us define our own, complicated, variable types. The `qsort` can even be used with those, since we get to define our own comparison function.

9.19. Conclusion

Writing your own functions in C is easy, and can be beneficial in several ways. Using functions can help you:

- Avoid duplicating the same code many times within a program.

  If you find yourself typing the same set of statements again and again, it's time to think about creating a function to replace them.

- Make your program easier to modify.

  After you've encapsulated a task within a function, you can easily modify it to make it better, without having to modify the rest of your program.

- Re-use your code in other programs.

  Once you've written your function, you can re-use it in other programs.

- Catch programming mistakes.

  The compiler makes some syntax checks when a function is called, so this is an opportunity to catch mistakes.

- Avoid accidentally changing variables.

  As we've seen, variables inside a function are independent from variables of the same name in other functions.

I encourage you to get into the habit of writing code that breaks work up into bite-sized functional chunks. Modularizing your programming jobs keeps you from reinventing solutions, and helps unclutter the visual flow of your programs, making it easier to see what the program is doing.

Later, we'll be learning how to create your own libraries of pre-compiled functions that you can reuse again and again.

Practice Problems

1. As we saw in Chapter 7, a Normal (or Gaussian) curve is described by the equation:

$$P(x) = Ae^{-\frac{(x-\overline{x})^2}{2s^2}}$$

If we let $A = 1$, $\overline{x} = 10$, and $s = 1$ the equation gets simplified to:

$$P(x) = e^{-\frac{(x-10)^2}{2}}$$

Write a program named `gauss.cpp` that contains a function named `P` that returns the value of $P(x)$ from the simplified equation above. Note that you'll need to include `math.h` at the top of your program so you can use the `exp` and `pow` functions. The function should take one `double` argument (the value of x) and return a `double` value.

In the `main` part of your program, create a loop that steps through 200 values of x, from zero to 19.9 in steps of 0.1. For each value of x, print x and $P(x)$.

If you put the program's output into a file and plot it with *gnuplot* you should see something like Figure 9.19.

2. Write your own version of Program 9.17 (`countto.cpp`) that adds a check to make sure the user has supplied a number on the command line. If the user doesn't give a number, the program should print a friendly message describing how to run the program, then exit without doing anything else. See Program 9.18 for an example that shows how to use the `exit` function.

3. Write a program named `sphere.cpp` that calculates the volume of a sphere, given its radius. Remember that the formula for the volume of a sphere is $V = \frac{4}{3}\pi r^3$. Use command-line arguments, as described in Section 9.15 above, to allow the user to specify the radius on the command line. For example, for a sphere of radius 3.5, the user should be able to run the program like this:

```
./sphere 3.5
```

The program should just print out the calculated volume with no commentary. So, if the sphere's radius is 3.5, the program should print `179.594380`.

Make sure the program checks to see if the user has specified the radius, and print an error message and exit if they haven't. See Program 9.18 for an example of this.

Hints: Since the radius can, in general, contain decimal places you'll need to use `atof` to convert the command-line argument to a number. See Program 9.18 for an example.

"Functional" is a solo piano piece composed by Thelonious Monk, "the genius of modern music".

Source: Wikimedia Commons

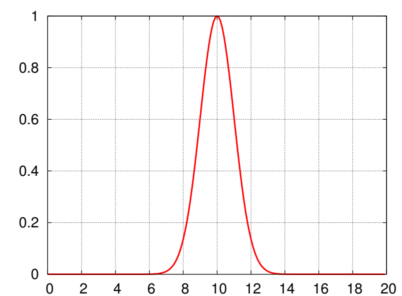


Figure 9.19: Your `gauss.cpp` program's output should look like this if you plot it with *gnuplot*.

The *Trylon* and *Perisphere* were two buildings made for the 1939 New York World's Fair. The Perisphere had a diameter of 180 feet. You could travel through it on a moving sidewalk and look down a diorama depicting a utopian city.

Source: Wikimedia Commons

4. Write a program named `compare.cpp` that contains a C function named "`similar`" that compares two `double` numbers and returns an integer value of 1 if the numbers differ by less than 0.0001, or 0 if they're farther apart. The program should ask the user for the two numbers to compare, and read them in with `scanf`. The program should use your function to compare the numbers, and tell the user (in clear, friendly words) if they're within 0.0001 of each other.

5. The formula for computing the amount of money in a savings account is:

$$M_{now} = M_{orig}\left(1 + \frac{r}{n}\right)^{nt}$$

where M_{now} is the amount of money you currently have, M_{orig} is the amount you originally deposited, r is the interest rate the bank is paying you, n is the number of times per year that the interest is added to your account, and t is the number of years since you originally deposited the money.

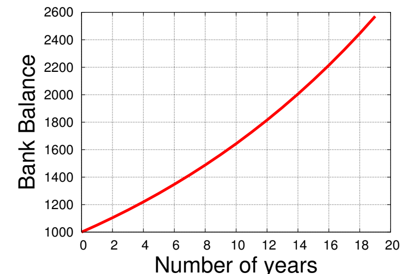


Figure 9.20: Bank balance over 20 years, starting with $1,000, with 5% interest compounded 4 times per year.

Write a program named `lucre.cpp` containing a function named `mnow` that begins like this:

```
double mnow ( double morig, double rate, int ntimes, int years )
```

where the arguments correspond to M_{orig}, r, n, and t, in that order. the function should use these arguments to compute M_{now}.

Your program should ask the user for the original amount of money in her/his account, then print how much money will be in the account each year for the next 20 years, assuming an interest rate of 0.05 (5%) with interest added 4 times per year. Use your `mnow` function to do the calculations. The program's output should be two columns: year number (0, 1, 2, ...*etc.*) and M_{now} for that year.

6. Write a program named `volumes.cpp` that calculates the volumes of some common shapes. Define three functions named `vsphere`, `vbox`, and `vcone` to calculate the volume of a sphere, a box, and a cone, respectively. Your definition of the vsphere function should start like this:

```
double vsphere ( double r )
```

where `r` is the sphere's radius. The definition of `vbox` should start like this:

```
double vbox ( double l, double w, double h )
```

where `l`, `w`, and `h` are the length, width, and height of the box. The definition of `vcone` should start like this:

```
double vcone ( double r, double h )
```

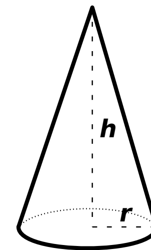


Figure 9.21: The volume of a **cone** is $\frac{1}{3}\pi r^2 h$, where r and h are as shown in the diagram above.
The volume of a **sphere** is $\frac{4}{3}\pi r^3$.
The volume of a **box** is just length × width × height.

where r is the radius of the cone's base and h is the cone's height. See Figure 9.21 for the formulas you'll need.

Your program should ask the user to specify which shape to use like this:

```
Enter the type of shape (1=sphere, 2=box, 3=cone):
```

then the program should ask the user for the dimensions of the shape, calculate its volume using the appropriate function, and tell the user the result.

7. Create a program that adds two numbers: Write a program named `add.cpp` that accepts two integers as command-line arguments (see Section 9.15 above). The program should add the two numbers and tell you what their sum is. For example, if you type this:

   ```
   ./add 23 52
   ```

   The program should print "75".

Speaking of adders, the harmless Eastern Hognosed Snake (*Heterodon platyrhinos*) is sometimes called a "puff adder" because it tries to frighten you by spreading its head like a cobra and hissing. If that doesn't work, it will roll over onto its back and play dead. If you turn it upright, it will roll over again just to prove that it's really dead. ("Don't bother me! I'm busy being dead!")

8. Write a program named `maxnum.cpp` that accepts a list of numbers on the command line and tells you which of the numbers is the largest. The program should accept numbers with decimal places, so you'll need to use `double` variables and the `atof` function. (See Section 9.15 above for information about using command-line arguments.) You should be able to run your program like this, for example:

   ```
   ./maxnum 12 13 128 765 2 4 3 -78
   Maximum number is 765.000000.
   ```

   Make sure your program can deal properly with negative numbers. If it's given the numbers -2 and -5, it should tell you that the largest number is -2.

9. Write a program named `testprime.cpp` that checks to see if a given integer is prime. (Remember that a prime number is one that can only be divided evenly by itself and 1.) The program should accept the number to be tested on the command line. For example:

   ```
   ./testprime 8675309
   ```

   The program should say something like "8675309 is prime" or "8675309 is NOT prime". The program should check the value of `argc` to make sure the user has supplied a number to be checked. If not, the program should tell the user what to do, and use `exit(1)` to stop. See Section 9.15 above for information about using command-line arguments.

   If we call the number to be checked n, then your program should look to see if n can be divided by any of the numbers from 2 to

Tommy Tutone, the band responsible for the 1981 hit song *867-5309/Jenny*.
Source: Wikimedia Commons

n-1, inclusive. You can use the % operator to check each number. Remember that n%i will be zero if n can be evenly divided by i. Refer to Chapter 4 for more information about the % or "modulo" operator.

Note that your program will only be able to work on numbers that are small enough to fit into an integer variable. On most computers, the biggest number that can fit into an int will be 2,147,483,647.

10. Write a new version of Program 9.6 (redbaron.cpp) that uses a different function for the flight path. Instead of the complicated function in Program 9.6, use:

$$h(x) = 10000 - \frac{(x-3000)^2}{10000}$$

and modify the "for" loop so that it does 6,000 steps instead of 1,000, tracking the plane over a distance of 6,000 meters.

Run your program and redirect the output into a file, then plot the file using *gnuplot*. What shape does it make? The graph should approximate the path followed by a "zero-G" aircraft near the top of its trajectory (see Figure 9.7).

Teacher and astronaut Christa Macauliffe experiencing weightlessness in NASA's "Vomit Comet". She died tragically in 1989, when the space shuttle *Challenger* exploded shortly after launch.

Source: Wikimedia Commons

11. In physics and math we often want to go through a list of numbers "cyclically". By this I mean that when we get to the end of the list we start back at the beginning again. For example, if our list contained the numbers 1, 2, 3 we could start at any of the numbers and write them down, in order, starting back at the beginning if necessary, until we'd written them all.

We could write this cyclic list in any of the following equivalent ways:

```
1 2 3
2 3 1
3 1 2
```

Notice that the list rotates in a particular direction, clockwise in this case, as shown in Figure 9.22.

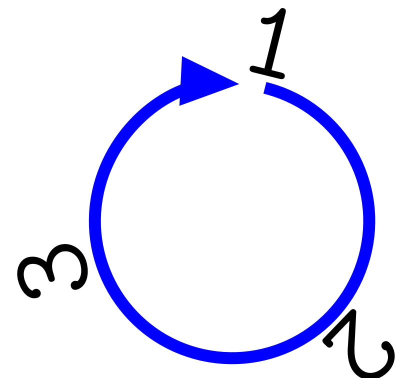


Figure 9.22: A cyclic list of numbers.

Imagine that we have three variables, i, j, and k with initial values i=1, j=2, and k=3. Write a function named rotate that uses the techniques described in Section 9.11 above to change the values of these variables, moving the value of i to j, the value of j to k, and the value of k to i. The function should start out like this:

```
void rotate ( int *i, int *j, int *k )
```

Use your function in a program named cycle.cpp that prints out the initial values of i, j, and k, then uses your rotate function to "rotate" the values of the three variables three times, printing out their new values after each rotation.

12. Sometimes a program needs to accept a file name on the command line. Write a program named `randfile.cpp` that can be run like this:

```
./randfile random.dat
```

When the program is run like this, it should generate 1,000 random numbers and write them into a new file named `random.dat`. You can just use the `rand` function to generate each random number.

The program should check to make sure the user has supplied a file name, and print an error message and exit if not. (See Program 9.18 for an example of this.)

Hint: The command-line arguments `argv[1]`, etc., are character strings, so you don't need to do any conversion with `atoi` or `atof`. In this program, you can just put `argv[1]` in place of the file name in your `fopen` statement.

The fascinating properties of strings.(*Sitzendes Mädchen mit einer Katze*, 1903, by Albert Anker.)
Source: WikiArt

13. A character string is just an array of characters. As we saw in Chapter 8, C provides us with a handy `strlen` function that can tell us the length of the text stored inside a character string. The `strlen` function does this by looking for the special "NUL" character that terminates the string.

Complete the following program (named `fakestrlen.cpp`) by adding a function named `mystrlen` that does the same thing the built-in `strlen` function does.

```
#include <stdio.h>
int mystrlen ( char string[] ) {
   // Insert your function here.
}
int main () {
  char string1[] = "Help, I'm trapped in a computer!";
  char string2[] = "Just kidding!";
  char string3[] = "They made me say that!";

  printf ( "String 1 length is %d\n", mystrlen( string1 ) );
  printf ( "String 2 length is %d\n", mystrlen( string2 ) );
  printf ( "String 3 length is %d\n", mystrlen( string3 ) );
}
```

Hints: Your function should use a `while` loop that starts with the first character of the string (character number zero) and checks each character to see if it's the NUL character, which is written as '\0' in C. The loop should continue for as long as the current character isn't a NUL. When the loop is done, the function should return the number of the current character.

14. Hot objects tend to emit heat and light in a range of wavelengths. The temperature of the object determines which wavelengths are emitted the most. In 1900 Max Planck wrote down the modern mathematical description of these emissions (known as "black body radiation"). The relationship between the intensity, I of radiation at a given wavelength, λ, depends on temperature, T, like this:

$$I(\lambda) = \frac{2hc^2}{\lambda^5} \frac{1}{\exp(\frac{hc}{\lambda kT}) - 1}$$

where **exp** is the exponential function and the physical constants are (in SI units):

Symbol	Name	Value
The speed of light in a vacuum	c	2.99792458×10^{8}
Planck's constant	h	$6.62606896 \times 10^{-34}$
Boltzmann's constant	k	$1.3806504 \times 10^{-23}$

Write a function named `intensity` that begins like this:

```
double intensity ( double lambda )
```

where `lambda` is λ and the function returns the value of $I(\lambda)$ from the equation above. Use a global variable named `t` to set the temperature to 5,000 Kelvin.

Use this function in a program named `planck.cpp`. The program should also contain the function named `plotit` that we used in Section 9.17. Have your program use the `plotit` function to print 100 values of $I(\lambda)$, with λ going from `0.1e-6` meters to `3e-6` meters. If you plot your results with *gnuplot* you should see a curve like the largest curve in Figure 9.23.

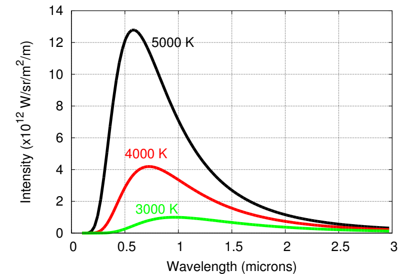


Figure 9.23: $I(\lambda)$ for several temperatures. Notice that the peak of I moves to the left as temperature increases. This shows that hotter objects emit more high-frequency radiation. (Low wavelengths correspond with high frequencies, and vice versa.)

15. Using the technique shown in Program 2.4 (diceroll.cpp) in Chapter 2, write a program that emulates a die with an abritrary number of sides. Call the new program `unidie.cpp`. The user should be able to specify the minimum and maximum numbers on the die by giving the program command-line arguments. The program should contain a function named `roll` that takes `min` and `max` as arguments and returns a random integer between `min` and `max`, inclusive. Make sure the program checks to see if the user has provided the necessary command-line arguments, and takes apprpriate action if not. When the program is run, it should print out the random number "rolled" by the die.

Detail from *Vanitas* by Adriaan Coorte.
Source: *Wikimedia Commons*