1. *Zero to Loops*

1.1. What's a Program?

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

The Antikythera Mechanism.
Source: Wikimedia Commons

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

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

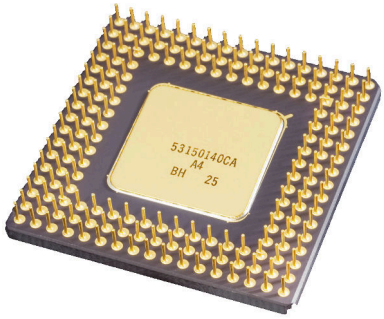
In the early 1800s, the English scientist and engineer Charles Babbage proposed a new kind of computer that he called an "Analytical Engine". This would be a general-purpose computer. Its behavior was controlled by punched cards (rectangular cards with a pattern of holes in them). By creating an appropriate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

Ada Lovelace, the first computer programmer.
Source: Wikimedia Commons

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

We distinguish between the computer's "hardware", which is fixed

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



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

Source: Wikimedia Commons

1.2. Creating Programs

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

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

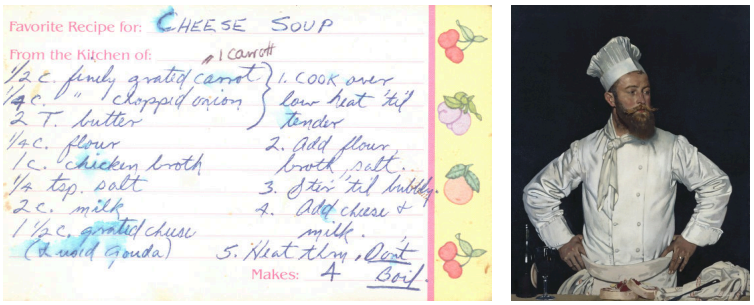


Figure 1.1: A program is just a recipe, but it needs to be translated into a language the computer can understand.

Source: Wikimedia Commons 1, 2

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

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

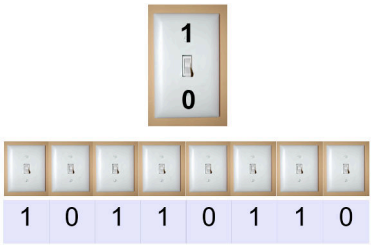


Figure 1.2: Bits as switches. You can think of each bit in a binary number as as switch. (In fact, programmers often talk about flipping bits on or off.) We group bits together in groups of eight because eight is a power of two (2^3), making it convenient for binary (base-2) arithmetic, just as 10, 100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

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

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

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

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

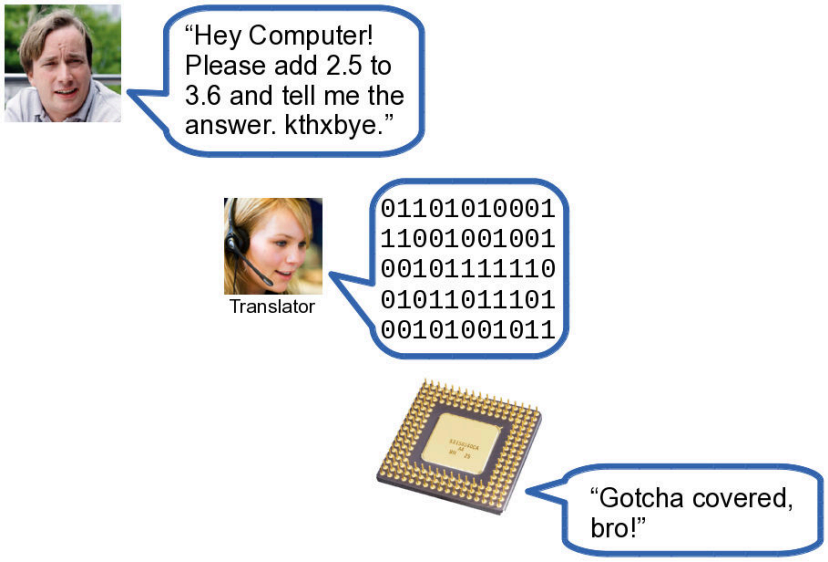


Figure 1.3: *Source: Wikimedia Commons 1, 2, 3*

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

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

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

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

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

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

Program 1.1: hello.cpp

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

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

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

At this point there are three obvious questions:

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

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

1.3. Files

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

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

[4] This description is often called the program's "source code"

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

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

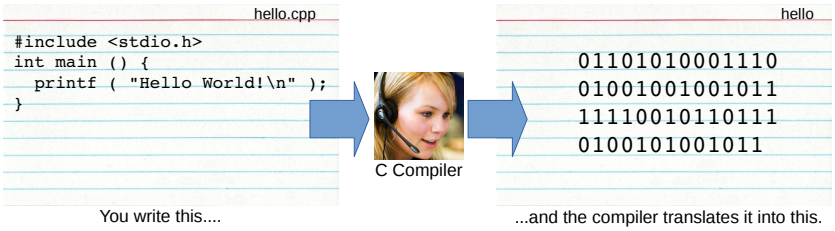


Figure 1.4: The C compiler reads our source code file and makes a binary file that the computer can understand.
Source: Wikimedia Commons

1.4. Your First Program

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

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

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

Writing a Program

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

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

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

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

Exercise 1: Creating a "Hello World" Program

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

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

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

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

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

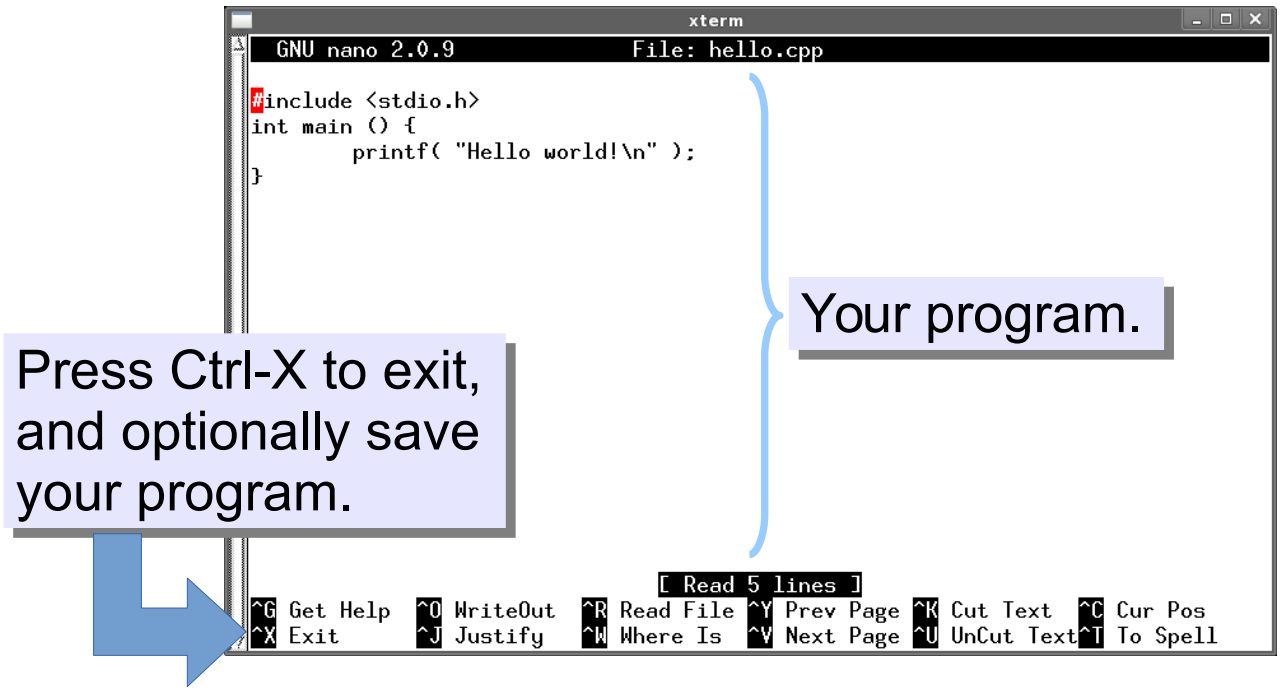


Figure 1.6: The editor called "*nano*".

Compiling Your Program

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

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

Exercise 2: Compiling "hello.cpp"

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

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

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

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

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

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

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

Running your program

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

Exercise 3: Run it!

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

```
./hello
```

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

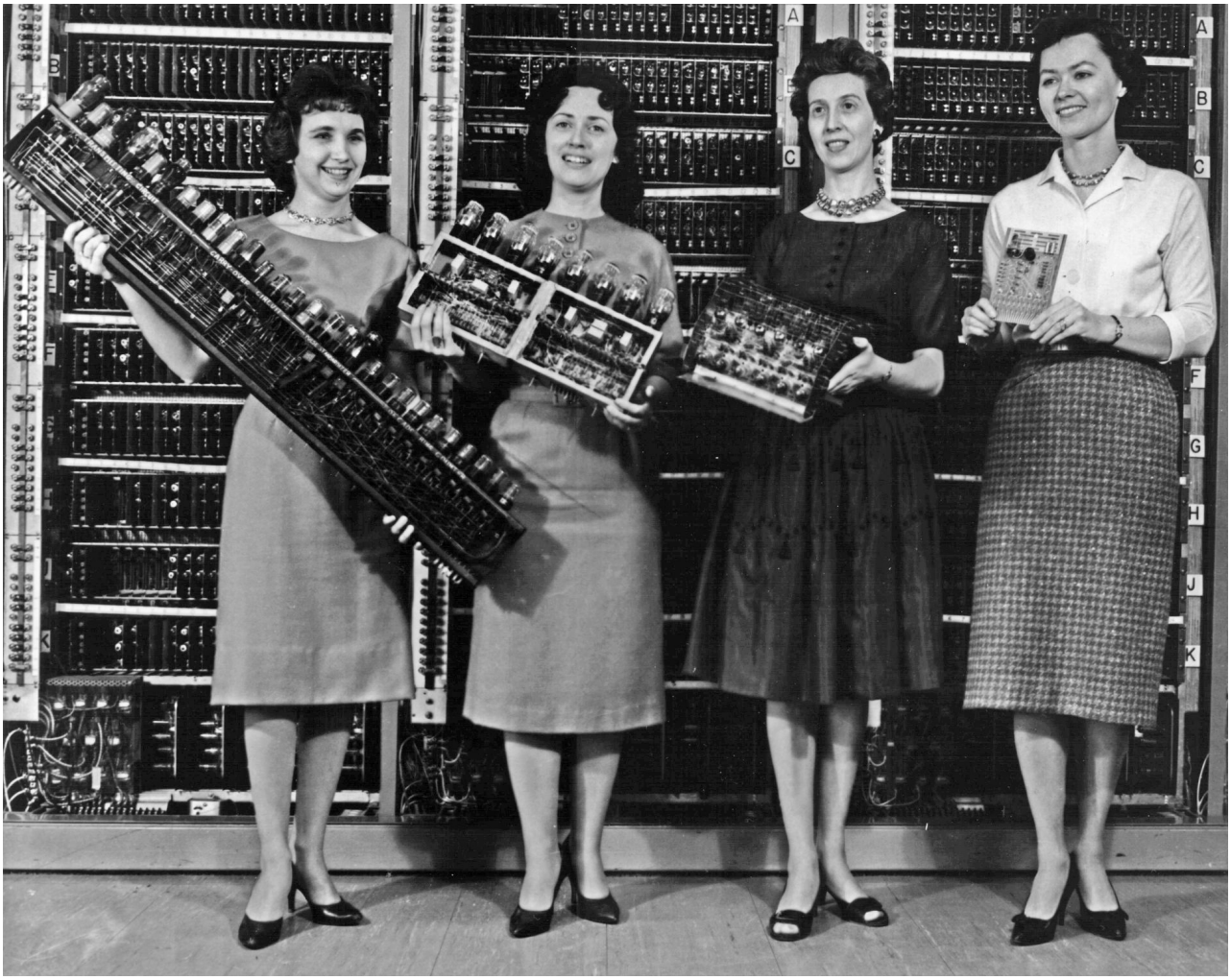


Figure 1.7: Congratulations!
Source: Wikimedia Commons

1.5. The Anatomy of a Program

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

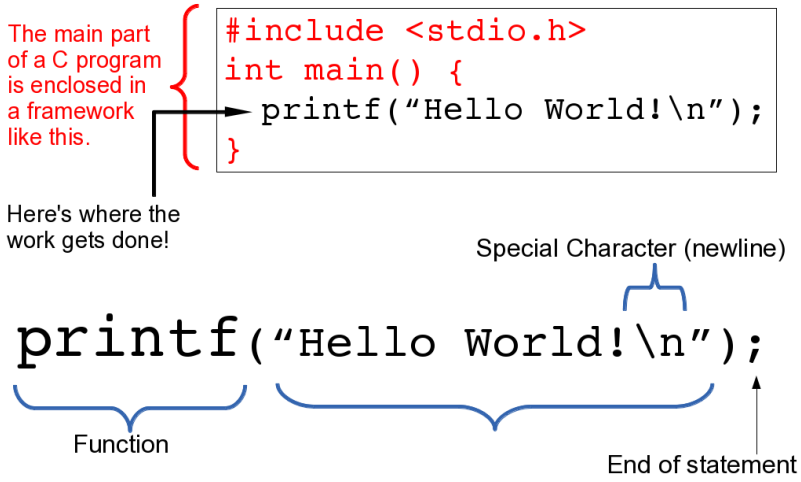


Figure 1.8: The anatomy of our "Hello World" program.

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

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

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

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

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

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

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

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

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

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

would print:

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

See the difference?

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

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

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

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

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

But what about...?

Could we write something like this?

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

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

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

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

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

any spaces after the "\", either.

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

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

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

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

1.6. Doing Math

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

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

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

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

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

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

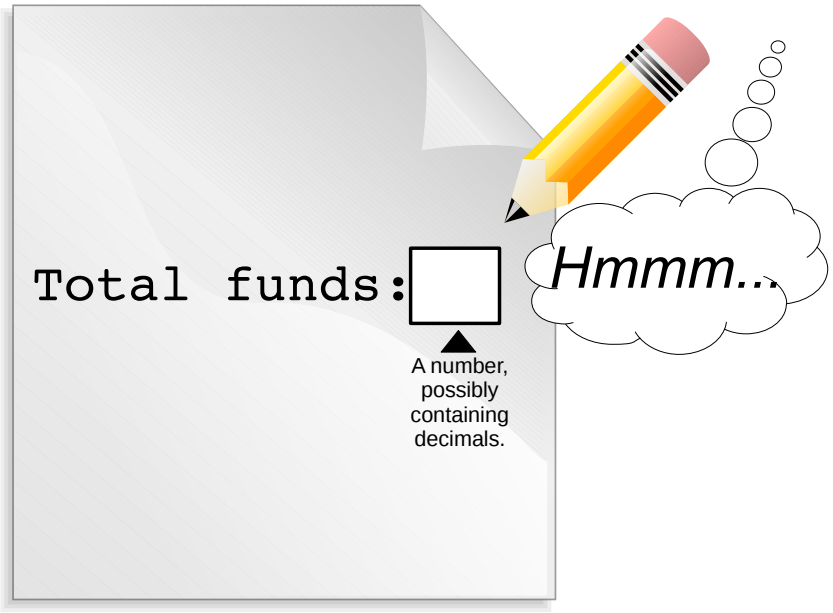


Figure 1.9: The text we give printf is like a fill-in form.

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

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

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

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

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

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

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

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

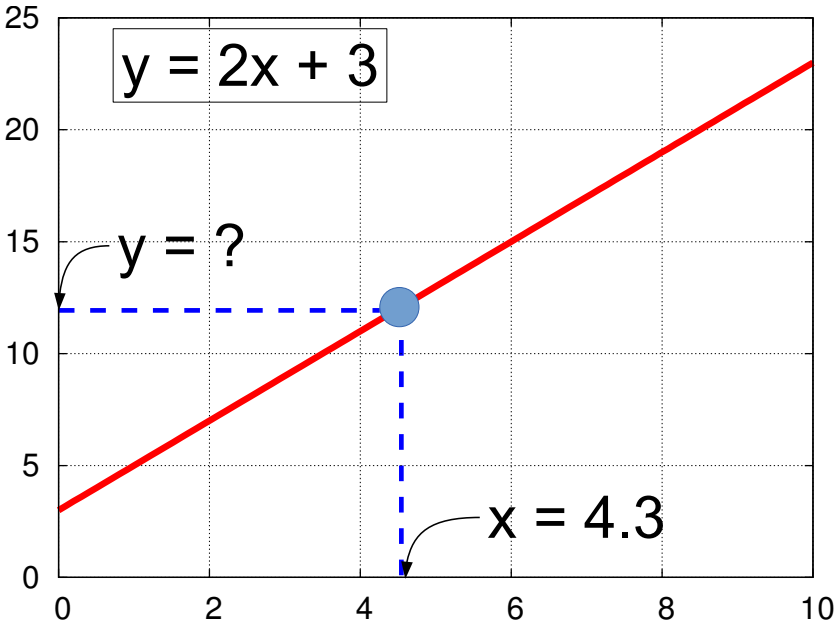


Figure 1.10: A line representing the equation y(x) = 2x + 3.

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

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

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

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

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

1.7. Variables

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

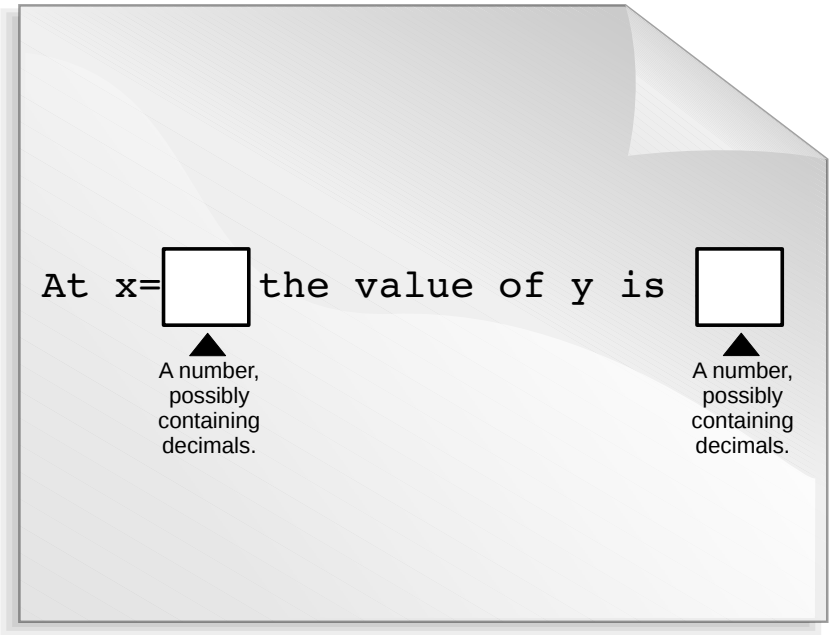


Figure 1.11: The `printf` text can contain more than one placeholder.

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

Here's another version of the program:

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

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

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

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

Definitions of Variables

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

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

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

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

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

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

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

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

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

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

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

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

Source: Wikimedia Commons

1.8. A Note About Algebra

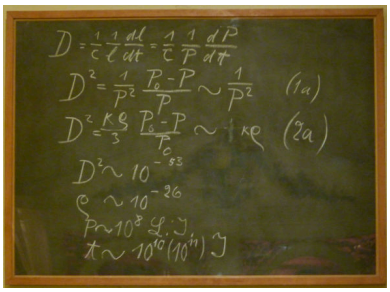


Figure 1.13: A blackboard used by Albert Einstein.
Source: Wikimedia Commons

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

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

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

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

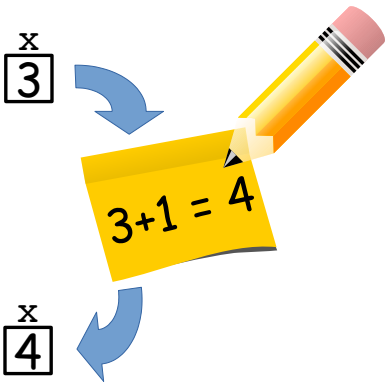


Figure 1.14: How the computer interprets the statement "`x = x + 1`". Remember that a variable in a C program is just a named storage location in the computer's memory. In this example, there's a variable named `x` that initially contains the value "3".

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

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

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

In algebra the statements "$y = 2x + 3$" and "$2x + 3 = y$" are equivalent, but not in C. Remember that a C program is like a recipe: it's a set of instructions that should be followed in a particular order. "Pour milk into a bowl" isn't the same as "pour bowl into a milk"! The latter doesn't make any sense, just as the statement "$2x + 3 = y$" wouldn't make sense in a C program.

1.9. Using Loops

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

We could, of course, do something like this:

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

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

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

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

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

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

Program 1.2: loop.cpp

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

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

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

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

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

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

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

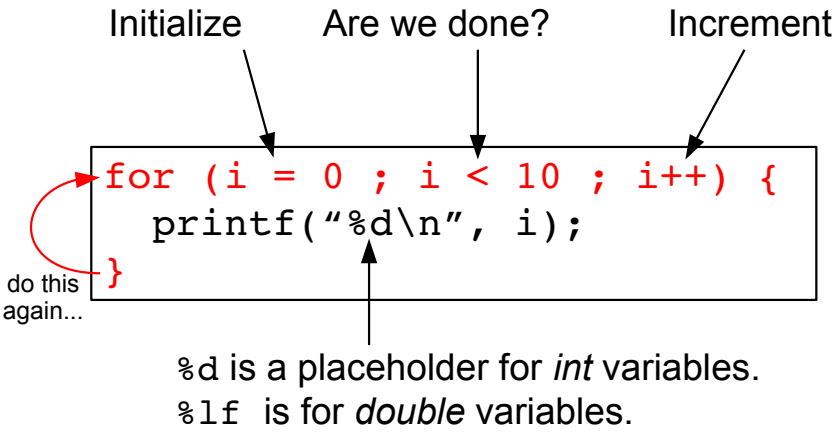


Figure 1.15: The anatomy of a "`for`" loop. The first line marks the top of the loop. The bottom line marks the end of the loop. Everything in between is done repeatedly, some number of times.

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

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

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

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

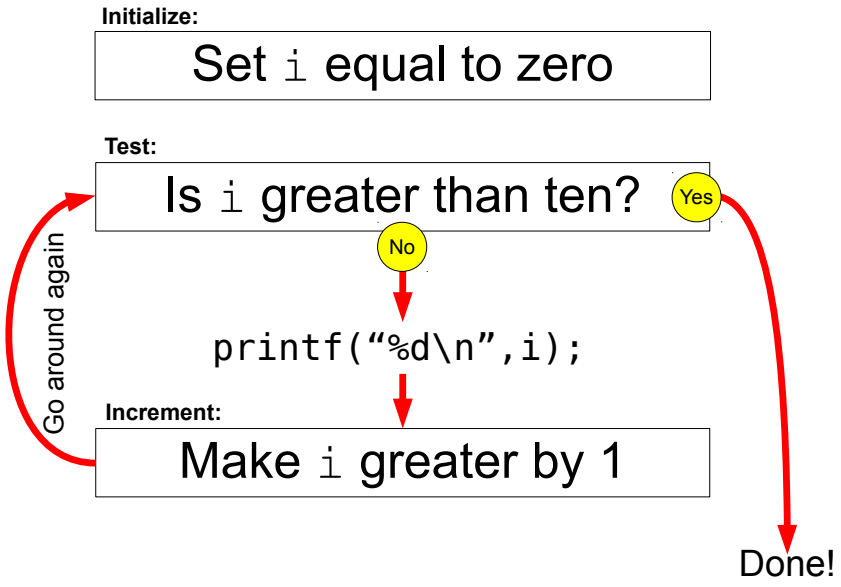


Figure 1.16: This diagram shows how a "`for`" loop works. Notice that if we gave `i` a value like 100 in the beginning, the program would never do the `printf`. Instead, it would just skip the loop entirely. This is important, because later on we'll encounter another kind of loop that will always be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

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

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

Exercise 4: Using Loops

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

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

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

But what about...?

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

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

1.10. Calculations Inside a Loop

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

Program 1.3: line.cpp

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

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

        int i;

        x = 0.0;

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

}
```

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

Exercise 5: Doing Math Inside a Loop

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

But what about...?

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

```
3.14159265358979323846264338327950288419716939
93751058209749445923078164062862089986280348 25
34211706798214808651328230664709384460955058 22
31725359408128481117450284102701938521105559 64
46229489549303819644288109756659334461284756 48
2337867831652712019091... et cetera
```

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

1.11. Graphing Our Results

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

Exercise 6: Making Graphs

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

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

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

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

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

gnuplot>
```

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

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

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

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

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

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

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

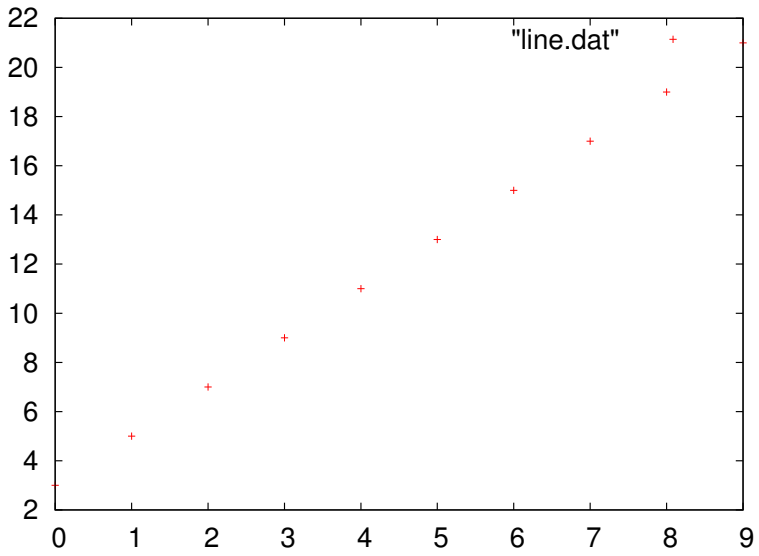


Figure 1.17: The result of typing `plot "line.dat"` in *gnuplot*.

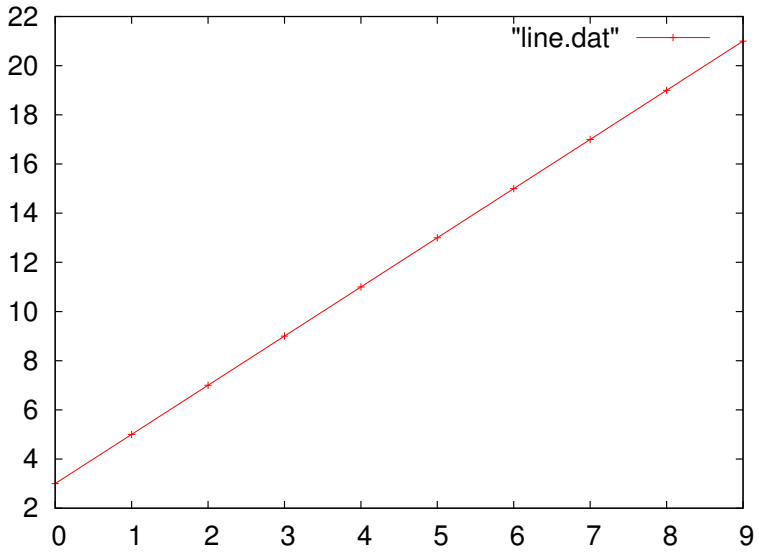


Figure 1.18: The result of typing `plot "line.dat" with linespoints` in *gnuplot*.

1.12. More About Variables

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

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

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

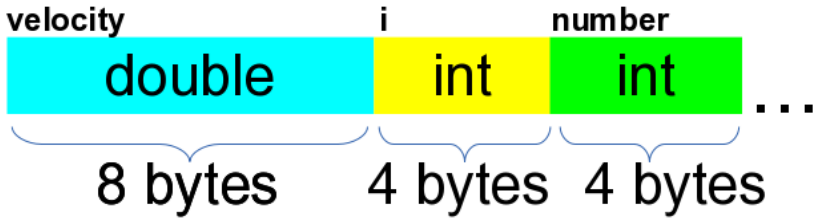


Figure 1.19: How a computer might store three variables in memory.

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

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

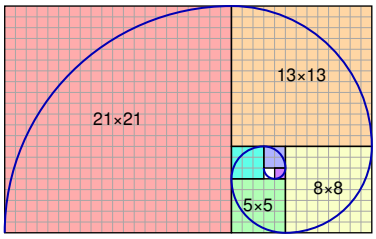
The C compiler tries to prevent this sort of thing two ways:

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

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

1.13. Fibonacci Numbers

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



A spiral made from squares whose sides are Fibonacci numbers.

Source: Wikimedia Commons

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

Program 1.4: fib.cpp

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

  int i;

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

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

    a = b;
    b = c;
  }

}
```

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

Print the first two numbers.

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

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

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

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

Fibonacci numbers, like this:

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

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

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

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

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

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

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

Source: Wikimedia Commons

Practice Problems

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

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

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

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

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

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

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

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

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

   Hint 1: Remember that you can use an arithmetic expression in a `printf` statement, as shown in Section 1.6.
   Hint 2: Remember that you can add or remove `\n` in a `printf`

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

Source: Wikimedia Commons

The Sherlock Holmes Museum at 221b Baker Street.

Source: Wikimedia Commons

statement to control whether it goes to the next line after printing some text, as shown in Section 1.5.

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

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

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

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

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

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

   Then start *gnuplot* and give it the command:

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

   The result should look like Figure 1.20.

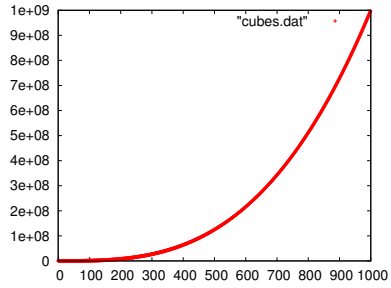


Figure 1.20: The output of your `cubes` program plotted by *gnuplot*.

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

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

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

```
-50.000000 1033.333333
-49.000000 1000.333333
-48.000000 968.000000
```

```
-47.000000 936.333333
-46.000000 905.333333
...
```

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

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

Then start *gnuplot* and give it the command:

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

The result should look like Figure 1.21.

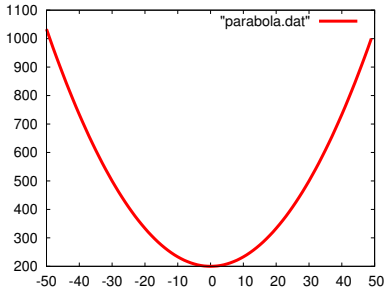


Figure 1.21: The output of your `curve` program plotted by *gnuplot*.

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

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

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

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

[16] See this Wikipedia article.