CHAPTER 2

First Steps in Programming

By now you're probably eager to create programs that allow your computer to really interact with the outside world. You don't just want programs that work as glorified typewriters, displaying fixed information that you included in the program code, and indeed there's a whole world of programming that goes beyond that.

Ideally, you want to be able to enter data from the keyboard and have the program squirrel it away somewhere. This would make the program much more versatile. Your program would be able to access and manipulate this data, and it would be able to work with different data values each time you execute it. This whole idea of entering different information each time you run a program is key to the whole enterprise of programming. A place to store an item of data that can vary in a program is not altogether surprisingly called a **variable**, and this is what this chapter covers.

This is quite a long chapter that covers a lot of ground. By the time you reach the end of it, you'll be able to write some really useful programs.

In this chapter you'll learn the following:

- How memory is used and what variables are
- How you can calculate in C
- What different types of variables there are and what you use them for
- What casting is and when you need to use it
- How to write a program that calculates the height of a tree—any tree

Memory in Your Computer

First let's look at how the computer stores the data that's processed in your program. To understand this, you need to know a little bit about memory in your computer, so before you go into your first program, let's take a quick tour of your computer's memory.

The instructions that make up your program, and the data that it acts upon, have to be stored somewhere while your computer is executing that program. When your program is running, this storage place is the machine's memory. It's also referred to as **main memory**, or the **random access memory** (**RAM**) of the machine.

Your computer also contains another kind of memory called **read-only memory** (**ROM**). As its name suggests, you can't change ROM: you can only read its contents or have your machine execute instructions contained within it. The information contained in ROM was put there when the machine was manufactured. This information is mainly programs that control the operation of the various devices attached to your computer, such as the display, the hard disk drive, the keyboard, and the floppy disk drive. On a PC, these programs are called the **basic input/output system** (**BIOS**) of your computer.

I don't need to refer to the BIOS in detail in this book. The interesting memory for your purposes is RAM; this is where your programs and data are stored when they execute. So let's learn a bit more about it.

You can think of your computer's RAM as an ordered sequence of boxes. Each of these boxes is in one of two states: either the box is full when it represents 1 or the box is empty when it represents 0. Therefore, each box represents one binary digit, either 0 or 1. The computer sometimes thinks of these in terms of **true** and **false**: 1 is true and 0 is false. Each of these boxes is called a **bit**, which is a contraction of *binary digit*.

Note If you can't remember or have never learned about binary numbers, and you want to find out a little bit more, you'll find more detail in Appendix A. However, you needn't worry about these details if they don't appeal to you. The important point here is that the computer can only deal with 1s and 0s—it can't deal with decimal numbers directly. All the data that your program works with, including the program instructions themselves, will consist of binary numbers internally.

For convenience, the boxes or bits in your computer are grouped into sets of eight, and each set of eight bits is called a **byte**. To allow you to refer to the contents of a particular byte, each byte has been labeled with a number, starting from 0 for the first byte, 1 for the second byte, and going up to whatever number of bytes you have in your computer's memory. This label for a byte is called its **address**. Thus, each byte will have an address that's different from that of all the other bytes in memory. Just as a street address identifies a particular house, the address of a byte uniquely references that byte in your computer's memory.

To summarize, you have your memory building blocks (called bits) that are in groups of eight (called bytes). A bit can only be either 1 or 0. This is illustrated in Figure 2-1.

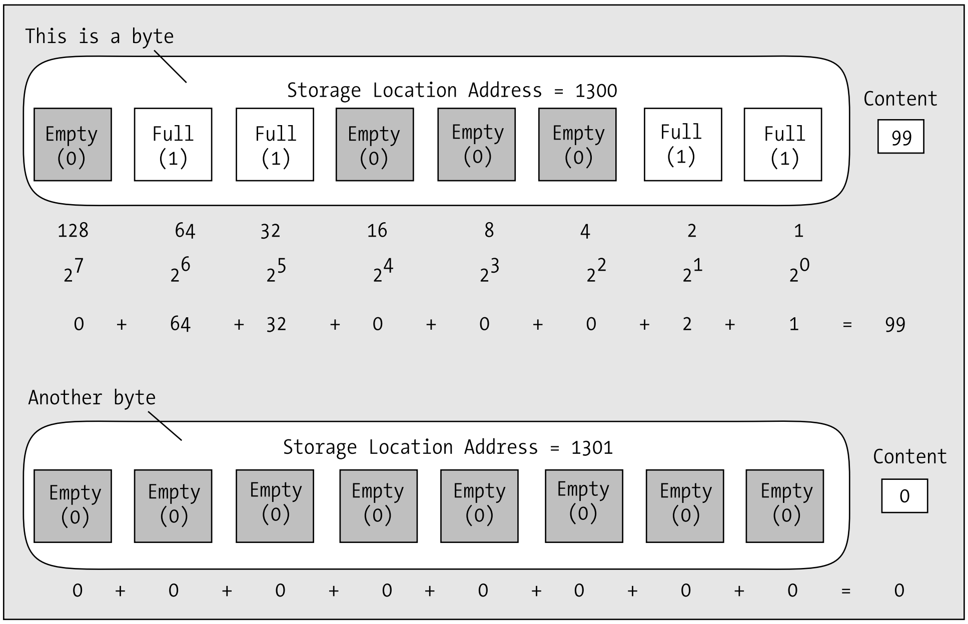

Figure 2-1. *Bytes in memory*

The amount of memory your computer has is expressed in terms of so many kilobytes, megabytes, or gigabytes. Here's what those words mean:

- 1 kilobyte (or 1KB) is 1,024 bytes.
- 1 megabyte (or 1MB) is 1,024 kilobytes, which is 1,048,576 bytes.
- 1 gigabyte (or 1GB) is 1,024 megabytes, which is 1,073,741,841 bytes.

You might be wondering why you don't work with simpler, more rounded numbers, such as a thousand, or a million, or a billion. The reason is this: there are 1,024 numbers from 0 to 1,023, and 1,023 happens to be 10 bits that are all 1 in binary: 11 1111 1111, which is a very convenient binary value. So while 1,000 is a very convenient decimal value, it's actually rather inconvenient in a binary machine—it's 11 1110 1000, which is not exactly neat and tidy. The kilobyte (1,024 bytes) is therefore defined in a manner that's convenient for your computer, rather than for you. Similarly, for a megabyte, you need 20 bits, and for a gigabyte, you need 30 bits. One point of confusion can arise here, particularly with disk drive capacities. Disk drive manufacturers often refer to a disk as having a capacity of 537 megabytes or 18.3 gigabytes, when they really mean 537 million bytes and 18.3 billion bytes. Of course, 537 million bytes is only 512 megabytes and 18.3 billion bytes is only 17 gigabytes, so a manufacturer's specification of the capacity of a hard disk can be misleading.

Now that you know a bit about bytes, let's see how you can use this memory in your programs.

What Is a Variable?

A variable is a specific piece of memory in your computer that consists of one or more contiguous bytes. Every variable has a name, and you can use that name to refer to that place in memory to retrieve what it contains or store a new data value there.

Let's start with a program that displays your salary using the `printf()` function that you saw in Chapter 1. Assuming your salary is $10,000 per month, you can already write that program very easily:

```
/* Program 2.1 What is a Variable? */
#include <stdio.h>

int main(void)
{
  printf("My salary is $10000");
  return 0;
}
```

I'm sure you don't need any more explanation about how this works; it's almost identical to the programs you developed in Chapter 1. So how can you modify this program to allow you to customize the message depending on a value stored in memory? There are, as ever, several ways of doing this. What they all have in common, though, is that they use a variable.

In this case, you could allocate a piece of memory that you could call, say, `salary`, and store the value `10000` in it. When you want to display your salary, you could use the name you've given to the variable, which is `salary`, and the value that's stored in it (`10000`) would be displayed. Wherever you use a variable name in a program, the computer accesses the value that's stored there. You can access a variable however many times you need to in your program. And when your salary changes, you can simply change the value stored in the variable `salary` and the whole program will carry on working with the new value. Of course, all these values will be stored as binary numbers inside the computer.

You can have as many variables as you like in a program. The value that each variable contains, at any point during the execution of that program, is determined by the instructions contained in your program. The value of a variable isn't fixed, and it can change as many times as you need it to throughout a program.

Variables That Store Numbers

There are several different types of variables, and each type of variable is used for storing a particular kind of data. You'll start by looking at variables that you can use to store numbers. There are actually several different ways in which you can store numbers in your program, so let's start with the simplest.

Integer Variables

Let's look first at variables that store integers. An integer is any whole number without a decimal point. Examples of integers are as follows:

1

10,999,000,000

1

You will recognize these values as integers, but what I've written here isn't quite correct so far as your program is concerned. You can't include commas in an integer, so the second value would actually be written in a program as 10999000000.

Here are some examples of numbers that are *not* integers:

1.234

999.9

2.0

–0.0005

Normally, 2.0 would be described as an integer because it's a whole number, but as far as your computer is concerned it isn't because it contains a decimal point. For your program, you must write it as 2 with no decimal point. In a C program integers are always written without a decimal point; if there's a decimal point, it isn't recognized as an integer. Before I discuss variables in more detail (and believe me, there's a lot more detail!), let's look at a simple variable in action in a program, just so you can get a feel for how they're used.

TRY IT OUT: USING A VARIABLE

Let's go back to your salary. You can try writing the previous program using a variable:

```
/* Program 2.2 Using a variable  */
#include <stdio.h>

int main(void)
{
  int salary;            /* Declare a variable called salary         */
  salary = 10000;        /* A simple arithmetic assignment statement */
  printf("My salary is %d.", salary);
  return 0;
}
```

Type in this example and compile, link, and execute it. You'll get the following output:

```
My salary is 10000.
```

How It Works

The first three lines are exactly the same as in all the previous programs. Let's look at the new stuff.

The statement that identifies the memory that you're using to store your salary is the following:

```
int salary;           /* Declare a variable called salary         */
```

This statement is called a **variable declaration** because it declares the name of the variable. The name, in this program, is `salary`.

Caution Notice that the variable declaration ends with a semicolon. If you omit the semicolon, your program will generate an error when you compile it.

The variable declaration also specifies the type of data that the variable will store. You've used the keyword `int` to specify that the variable, `salary`, will be used to store an integer value. The keyword `int` precedes the name of the variable. As you'll see later, declarations for variables that store other kinds of data consist of another keyword specifying a data type followed by a variable name in a similar manner.

Note Remember, keywords are special C words that mean something specific to the compiler. You must not use them as variable names or your compiler will get confused.

The variable declaration is also a **definition** for the variable, `salary`, because it causes some storage to be allocated to store an integer value that can be referred to using the name `salary`. Of course, you have not specified what the value of salary should be yet, so at this point it will contain a junk value—whatever was left behind from when this bit of memory was used last.

The next statement is the following:

```
salary = 10000;
```

This is a simple **arithmetic assignment statement**. It takes the value to the right of the equal sign and stores it in the variable on the left of the equal sign. Here you're declaring that the variable `salary` will have the value `10000`. You're storing the value on the right (`10000`) in the variable on the left (`salary`). The `=` symbol is called the **assignment operator** because it assigns the value on the right to the variable on the left.

You then have the familiar `printf()` statement, but it's a little different from how you've seen it in action before:

```
printf("My salary is %d.", salary);
```

There are now two **arguments** inside the parentheses, separated by a comma. An argument is a value that's passed to a function. In this program statement, the two arguments to the `printf()` function are as follows:

- Argument 1 is a **control string**, so called because it controls how the output specified by the following argument (or arguments) is to be presented. This is the character string between the double quotes. It is also referred to as a **format string** because it specifies the format of the data that is output.
- Argument 2 is the name of the variable `salary`. How the value of this variable will be displayed is determined by the first argument—the control string.

The control string is fairly similar to the previous example, in that it contains some text to be displayed. However, if you look carefully, you'll see `%d` embedded in it. This is called a **conversion specifier** for the variable.

Conversion specifiers determine how variables are displayed on the screen. In this case, you've used a `d`, which is a decimal specifier that applies to integer values (whole numbers). It just means that the second argument, `salary`, will be interpreted and output as a decimal (base 10) number.

Note Conversion specifiers always start with a % character so that the `printf()` function can recognize them. Because a % in a control string always indicates the start of a conversion specifier, if you want to output a % character you must use the sequence %%.

TRY IT OUT: USING MORE VARIABLES

Let's try a slightly larger example:

```
/* Program 2.3 Using more variables */
#include <stdio.h>

int main(void)
{
  int brothers;               /* Declare a variable called brothers */
  int brides;                 /* and a variable called brides       */

  brothers = 7;               /* Store 7 in the variable brothers   */
  brides = 7;                 /* Store 7 in the variable brides     */

  /* Display some output */
  printf("%d brides for %d brothers", brides, brothers);
  return 0;
}
```

If you run this program you should get the following output:

```
7 brides for 7 brothers
```

How It Works

This program works in a very similar way to the previous example. You first declare two variables, `brides` and `brothers`, with the following statements:

```
  int brothers;               /* Declare a variable called brothers */
  int brides;                 /* and a variable called brides       */
```

Both of these variables are declared as type `int` so they both store integer values. Notice that they've been declared in separate statements. Because they're both of the same type, you could have saved a line of code and declared them together like this:

```
int brothers, brides;
```

When you declare several variables in one statement, the variable names following the data type are separated by commas, and the whole line ends with a semicolon. This can be a convenient format, although there's a downside in that it isn't so obvious what each variable is for, because if they appear on a single line you can't add individual comments to describe each variable. However, you could write this single statement spread over two lines:

```
int brothers,        /* Declare a variable called brothers */
     brides;         /* and a variable called brides       */
```

By spreading the statement out over two lines, you're able to put the comments back in. The comments will be ignored by the compiler, so it's still the exact equivalent of the original statement without the comments. Of course, you might as well write it as two statements.

Note that the declarations appear at the beginning of the executable code for the function. You should put all the declarations for variables that you intend to use at the beginning.

The next two statements assign the same value, 7, to each of the variables:

```
brothers = 7;              /* Store 7 in the variable brothers   */
brides = 7;                /* Store 7 in the variable brides     */
```

Note that the statements that declared these variables precede these statements. If one or other of the declarations were missing or appeared later in the code, the program wouldn't compile.

The next statement calls the `printf()` function with a control string as the first argument that will display a line of text. The `%d` conversion specifiers within this control string will be replaced by the values currently stored in the variables that appear as the second and third arguments to the `printf()` function call—in this case, `brides` and `brothers`:

```
printf("%d brides for %d brothers", brides, brothers);
```

The conversion specifiers are replaced in order by the values of the variables that appear as the second and subsequent arguments to the `printf()` function, so the value of `brides` corresponds to the first specifier, and the value of `brothers` corresponds to the second. This would be more obvious if you changed the statements that set the values of the variables as follows:

```
brothers = 8;              /* Store 8 in the variable brothers   */
brides = 4;                /* Store 4 in the variable brides     */
```

In this somewhat dubious scenario, the `printf()` statement would show clearly which variable corresponds to which conversion specifier, because the output would be the following:

```
4 brides for 8 brothers
```

Naming Variables

The name that you give to a variable, conveniently referred to as a **variable name**, can be defined with some flexibility. A variable name is a sequence of one or more uppercase or lowercase letters, digits, and underscore characters (_) that begins with a letter (incidentally, the underscore character counts as a letter). Examples of legal variable names are as follows:

Radius

diameter

Auntie_May

Knotted_Wool

D678

Because a variable name can't begin with a digit, 8_Ball and 6_pack aren't legal names. A variable name can't include any other characters besides letters, underscores, and digits, so Hash! and Mary-Lou aren't allowed as names. This last example is a common mistake, but Mary_Lou would be quite acceptable. Because spaces aren't allowed in a name, Mary Lou would be interpreted as two variable names, Mary and Lou. Variables starting with one or two underscore characters are often used in the header files, so don't use the underscore as the first letter when naming your variables; otherwise, you run the risk of your name clashing with the name of a variable used in the standard library. For example, names such as _this and _that are best avoided.

Although you can call variables whatever you want within the preceding constraints, it's worth calling them something that gives you a clue to what they contain. Assigning the name x to a variable that stores a salary isn't very helpful. It would be far better to call it salary and leave no one in any doubt as to what it is.

Caution The number of characters that you can have in a variable name will depend upon your compiler. A minimum of 31 characters must be supported by a compiler that conforms to the C language standard, so you can always use names up to this length without any problems. I suggest that you don't make your variable names longer than this anyway, as they become cumbersome and make the code harder to follow. Some compilers will truncate names that are too long.

Another very important point to remember when naming your variables is that variable names are case sensitive, which means that the names Democrat and democrat are distinct. You can demonstrate this by changing the printf() statement so that one of the variable names starts with a capital letter, as follows:

```
/* Program 2.3A Using more variables */
#include <stdio.h>

int main(void)
{
  int brothers;              /* Declare a variable called brothers */
  int brides;                /* and a variable called brides       */

  brothers = 7;              /* Store 7 in the variable brothers   */
  brides = 7;                /* Store 7 in the variable brides     */

  /* Display some output */
  printf("%d brides for %d brothers", Brides, brothers);
  return 0;
}
```

You'll get an error message when you try to compile this version of the program. The compiler interprets the two variable names brides and Brides as different, so it doesn't understand what Brides refers to. This is a common error. As I've said before, punctuation and spelling mistakes are one of the main causes of trivial errors.

You must also declare a variable before you use it, otherwise the compiler will not recognize it and will flag the statement as an error.

Using Variables

You now know how to name and declare your variables, but so far this hasn't been much more useful than anything you learned in Chapter 1. Let's try another program in which you'll use the values in the variables before you produce the output.

TRY IT OUT: DOING A SIMPLE CALCULATION

This program does a simple calculation using the values of the variables:

```
/* Program 2.4 Simple calculations */
#include <stdio.h>

int main(void)
{
  int Total_Pets;
  int Cats;
  int Dogs;
  int Ponies;
  int Others;

  /* Set the number of each kind of pet */
  Cats = 2;
  Dogs = 1;
  Ponies = 1;
  Others = 46;

  /* Calculate the total number of pets */
  Total_Pets = Cats + Dogs + Ponies + Others;

  printf("We have %d pets in total", Total_Pets);  /* Output the result */
  return 0;
}
```

This example produces this output:

```
We have 50 pets in total
```

How It Works

As in the previous examples, all the statements between the braces are indented by the same amount. This makes it clear that all these statements belong together. You should always organize your programs the way you see here: indent a group of statements that lie between an opening and closing brace by the same amount. It makes your programs much easier to read.

You first define five variables of type `int`:

```
int Total_Pets;
int Cats;
int Dogs;
int Ponies;
int Others;
```

Because each of these variables will be used to store a count of a number of animals, they are definitely going to be whole numbers. As you can see, they're all declared as type `int`.

Note that you could have declared all five variables in a single statement and include comments, as follows:

```
int Total_Pets,                  /* The total number of pets     */
    Cats,                        /* The number of cats as pets   */
    Dogs,                        /* The number of dogs as pets   */
    Ponies,                      /* The number of ponies as pets */
    Others;                      /* The number of other pets     */
```

These are rather superfluous comments but they illustrate the point. The statement is spread over several lines so that you can add the comments in an orderly fashion. Notice that there are commas separating each of the variable names. Because the comments are ignored by the compiler, this is exactly the same as the following statement:

```
int Total_Pets, Cats, Dogs, Ponies, Others;
```

You can spread C statements over as many lines as you want. The semicolon determines the end of the statement, not the end of the line.

Now back to the program. The variables are given specific values in these four assignment statements:

```
Cats = 2;
Dogs = 1;
Ponies = 1;
Others = 46;
```

At this point the variable `Total_Pets` doesn't have an explicit value set. It will get its value as a result of the calculation using the other variables:

```
Total_Pets = Cats + Dogs + Ponies + Others;
```

In this arithmetic statement, you calculate the sum of all your pets on the right of the assignment operator by adding the values of each of the variables together. This total value is then stored in the variable `Total_Pets` that appears on the left of the assignment operator. The new value replaces any old value that was stored in the variable `Total_Pets`.

The `printf()` statement presents the result of the calculation by displaying the value of `Total_Pets`:

```
printf("We have %d pets in total", Total_Pets);
```

Try changing the numbers of some of the types of animals, or maybe add some more of your own. Remember to declare them, initialize their value, and include them in the `Total_Pets` statement.

Initializing Variables

In the previous example, you declared each variable with a statement such as this:

```
int Cats;                        /* The number of cats as pets   */
```

You set the value of the variable `Cats` using this statement:

```
Cats = 2;
```

This sets the value of the variable `Cats` to 2.

So what was the value before this statement was executed? Well, it could be anything. The first statement creates the variable called `Cats`, but its value will be whatever was left in memory from the last program that used this bit of memory. The assignment statement that appeared later set the value to 2, but it would be much better to initialize the variable when you declare it. You can do this with the following statement:

```
int Cats = 2;
```

This statement declares the variable `Cats` as type `int` *and* sets its initial value to 2.

Initializing variables as you declare them is a very good idea in general. It avoids any doubt about what the initial values are, and if the program doesn't work as it should, it can help you track down the errors. Avoiding leaving spurious values for variables when you create them also reduces the chances of your computer crashing when things do go wrong. Inadvertently working with junk values can cause all kinds of problems. From now on, you'll always initialize variables in the examples, even if it's just to 0.

Arithmetic Statements

The previous program is the first one that really did something. It is very simple—just adding a few numbers—but it is a significant step forward. It is an elementary example of using an arithmetic statement to perform a calculation. Now let's look at some more sophisticated calculations that you can do.

Basic Arithmetic Operations

In C, an arithmetic statement is of the following form:

```
Variable_Name = Arithmetic_Expression;
```

The arithmetic expression on the right of the `=` operator specifies a calculation using values stored in variables and/or explicit numbers that are combined using arithmetic operators such as addition (`+`), subtraction (`–`), multiplication (`*`), and division (`/`). There are also other operators you can use in an arithmetic expression, as you'll see.

In the previous example, the arithmetic statement was the following:

```
Total_Pets = Cats + Dogs + Ponies + Others;
```

The effect of this statement is to calculate the value of the arithmetic expression to the right of the `=` and store that value in the variable specified on the left.

In C, the `=` symbol defines an action. It doesn't specify that the two sides are equal, as it does in mathematics. It specifies that the value resulting from the expression on the right is to be stored in the variable on the left. This means that you could have the following:

```
Total_Pets = Total_Pets + 2;
```

This would be ridiculous as a mathematical equation, but in programming it's fine. Let's look at it in context. Imagine you'd rewritten the last part of the program to include the preceding statement. Here's a fragment of the program as it would appear with the statement added:

```
Total_Pets = Cats + Dogs + Ponies + Others;
Total_Pets = Total_Pets + 2;
printf("The total number of pets is: %d", Total_Pets);
```

After executing the first statement here, `Total_Pets` will contain the value 50. Then, in the second line, you extract the value of `Total_Pets`, add 2 to that value and store the result back in the variable `Total_Pets`. The final total that will be displayed is therefore 52.

■**Note** In assignment operations, the expression on the right side of the = sign is evaluated first, and the result is then stored in the variable on the left. The new value replaces the value that was previously contained in the variable to the left of the assignment operator. The variable on the left of the assignment is called an `lvalue`, because it is a location that can store a value. The value that results from executing the expression on the right of the assignment is called an `rvalue` because it is simply a value that results from evaluating the expression.

Any expression that results in a numeric value is described as an **arithmetic expression**. The following are arithmetic expressions:

```
3

1 + 2

Total_Pets

Cats + Dogs - Ponies
```

Evaluating any of these expressions produces a single numeric value. Note that just a variable name is an expression that is evaluated to produce a value: the value that the variable contains. In a moment, you'll take a closer look at how an expression is made up, and you'll look into the rules governing its evaluation. First, though, you'll try some simple examples using the basic arithmetic operators that you have at your disposal. Table 2-1 shows these operators.

Table 2-1. *Basic Arithmetic Operators*

Operator	Action
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

You may not have come across the **modulus operator** before. It just calculates the remainder after dividing the value of the expression on the left of the operator by the value of the expression on the right. For this reason it's sometimes referred to as the **remainder operator**. The expression 12 % 5 would produce 2, because 12 divided by 5 leaves a remainder of 2. You'll look at this in more detail in the next section, "More on Division with Integers." All these operators work as you'd expect, with the exception of division, which is slightly nonintuitive when applied to integers, as you'll see. Let's try some more arithmetic operations.

TRY IT OUT: SUBTRACTION AND MULTIPLICATION

Let's look at a food-based program that demonstrates subtraction and multiplication:

```
/* Program 2.5 Calculations with cookies */
#include <stdio.h>

int main(void)
{
  int cookies = 5;
  int cookie_calories = 125;    /* Calories per cookie */
  int total_eaten = 0;          /* Total cookies eaten */

  int eaten = 2;                /* Number to be eaten  */
  cookies = cookies - eaten;    /* Subtract number eaten from cookies */
  total_eaten = total_eaten + eaten;
  printf("\nI have eaten %d cookies.  There are %d cookies left",
                                                       eaten, cookies);

  eaten = 3;                       /* New value for cookies to be eaten  */
  cookies = cookies - eaten;       /* Subtract number eaten from cookies */
  total_eaten = total_eaten + eaten;
  printf("\nI have eaten %d more.  Now there are %d cookies left\n",
                                                       eaten, cookies);
  printf("\nTotal energy consumed is %d calories.\n",
                                         total_eaten*cookie_calories);
  return 0;
}
```

This program produces the following output:

```
I have eaten 2 cookies.  There are 3 cookies left
I have eaten three more.  Now there are 0 cookies left

Total energy consumed is 625 calories.
```

How It Works

You first declare and initialize three variables of type `int`:

```
  int cookies = 5;
  int cookie_calories = 125;       /* Calories per cookie */
  int total_eaten = 0;             /* Total cookies eaten */
```

You'll use the `total_eaten` variable to accumulate the total number of cookies eaten as execution of the program progresses, so you initialize it to 0.

The next variable that you declare and initialize holds the number of cookies to be eaten next:

```
  int eaten = 2;                   /* Number to be eaten */
```

You use the subtraction operator to subtract `eaten` from the value of `cookies`:

```
  cookies = cookies - eaten;       /* Subtract number eaten from cookies */
```

The result of the subtraction is stored back in the variable cookies, so the value of cookies will now be 3.

Because you've eaten some cookies, you increment the count of the total that you've eaten by the value of eaten:

```
total_eaten = total_eaten + eaten;
```

You add the current value of eaten, which is 2, to the current value of total_eaten, which is 0. The result is stored back in the variable total_eaten.

The printf() statement displays the number of cookies that are left:

```
printf("\nI have eaten %d cookies.  There are %d cookies left",
                                                      eaten, cookies);
```

I couldn't fit the statement in the space available, so after the comma following the first argument to printf(), I put the rest of the statement on a new line. You can spread statements out like this to make them easier to read or fit within a given width on the screen.

Note that you *cannot* split the string that is the first argument in this way. An explicit newline character isn't allowed in the middle of a string. When you need to split a string over two or more lines, each segment of the string on a line must have its own pair of double quotes delimiting it. For example, you could write the previous statement as follows:

```
printf("\nI have eaten %d cookies. "
                    " There are %d cookies left",
                                                      eaten, cookies);
```

Where there are two or more strings immediately following one another like this, the compiler will join them together to form a single string.

You display the values stored in eaten and cookies using the conversion specifier, %d, for integer values. The value of eaten will replace the first %d in the output string, and the value of cookies will replace the second. The string will be displayed starting on a new line because of the \n at the beginning.

The next statement sets the variable eaten to a new value:

```
eaten = 3;                        /* New value for cookies to be eaten  */
```

The new value replaces the previous value stored in eaten, which was 2. You then go through the same sequence of operations as you did before:

```
cookies = cookies - eaten;       /* Subtract number eaten from cookies */
total_eaten = total_eaten + eaten;
printf("\nI have eaten %d more.  Now there are %d cookies left\n",
                                                      eaten, cookies);
```

Finally, before executing the return statement that ends the program, you calculate and output the number of calories corresponding to the number of cookies eaten:

```
printf("\nTotal energy consumed is %d calories.\n",
                                         total_eaten*cookie_calories);
```

Here the second argument to the printf() function is an arithmetic expression rather than just a variable. The compiler will arrange for the result of the expression total_eaten*cookie_calories to be stored in a temporary variable, and that value will be passed as the second argument to the printf() function. You can always use an expression for an argument to a function as long as it evaluates to a result of the required type.

Easy, isn't it? Let's take a look at division and the modulus operator.

TRY IT OUT: DIVISION AND THE MODULUS OPERATOR

Suppose you have a jar of 45 cookies and a group of seven children. You'll divide the cookies equally among the children and work out how many each child has. Then you'll work out how many cookies are left over.

```
/* Program 2.6 Cookies and kids */
#include <stdio.h>

int main(void)
{
  int cookies = 45;                     /* Number of cookies in the jar */
  int children = 7;                     /* Number of children           */
  int cookies_per_child = 0;            /* Number of cookies per child  */
  int cookies_left_over = 0;            /* Number of cookies left over  */

  /* Calculate how many cookies each child gets when they are divided up */
  cookies_per_child = cookies/children;  /* Number of cookies per child  */
  printf("You have %d children and %d cookies", children, cookies);
  printf("\nGive each child %d cookies.", cookies_per_child);

  /* Calculate how many cookies are left over */
  cookies_left_over = cookies%children;
  printf("\nThere are %d cookies left over.\n", cookies_left_over);
  return 0;
}
```

When you run this program you'll get this output:

```
You have 7 children and 45 cookies
Give each child 6 cookies.
There are 3 cookies left over.
```

How It Works

Let's go through this program step by step. Four integer variables, cookies, children, cookies_per_child, and cookies_left_over are declared and initialized with the following statements:

```
  int cookies = 45;                     /* Number of cookies in the jar */
  int children = 7;                     /* Number of children           */
  int cookies_per_child = 0;            /* Number of cookies per child  */
  int cookies_left_over = 0;            /* Number of cookies left over  */
```

The number of cookies is divided by the number of children by using the division operator / to produce the number of cookies given to each child:

```
  cookies_per_child = cookies/children;  /* Number of cookies per child */
```

The next two statements output what is happening, including the value stored in cookies_per_child:

```
  printf("You have %d children and %d cookies", children, cookies);
  printf("\nGive each child %d cookies.", cookies_per_child);
```

You can see from the output that `cookies_per_child` has the value 6. This is because the division operator always produces an integer result when the operands are integers. The result of dividing 45 by 7 is 6, with a remainder of 3. You calculate the remainder in the next statement by using the modulus operator:

```
cookies_left_over = cookies%children;
```

The expression to the right of the assignment operator calculates the remainder that results when the value of `cookies` is divided by the value of `children`.

Finally, you output the reminder in the last statement:

```
printf("\nThere are %d cookies left over.\n", cookies_left_over);
```

More on Division with Integers

Let's look at the result of using the division and modulus operators where one or other of the operands is negative. With division, if the operands have different signs, the result will be negative. Thus, the expression –45 / 7 produces the same result as the expression 45 / –7, which is –6. If the operands in a division are of the same sign, positive or negative, the result is positive. Thus, 45 / 7 produces the same result as –45 / –7, which is 6.

With the modulus operator, the sign of the result is always the same as the sign of the left operand. Thus, 45 % –7 results in the value 3, whereas –45 % 7 results in the value –3.

Unary Operators

The operators that you've dealt with so far have been **binary operators**. These operators are called binary operators because they operate on *two* data items. Incidentally, the items of data that an operator applies to are generally referred to as **operands**. For example, the multiplication is a binary operator because it has two operands and the effect is to multiply one operand value by the other. However, there are some operators that are unary, meaning that they only need one operand. You'll see more examples later, but for now you'll just take a look at the single most common unary operator.

The Unary Minus Operator

You'll find the unary operator very useful in more complicated programs. It makes whatever is positive negative, and vice versa. You might not immediately realize when you would use this, but think about keeping track of your bank account. Say you have $200 in the bank. You record what happens to this money in a book with two columns, one for money that you pay out and another for money that you receive. One column is your expenditure (negative) and the other is your revenue (positive).

You decide to buy a CD for $50 and a book for $25. If all goes well, when you compare the initial value in the bank and subtract the expenditure ($75), you should end up with what's left. Table 2-2 shows how these entries could typically be recorded.

Table 2-2. *Recording Revenues and Expenditures*

Entry	Revenue	Expenditure	Bank Balance
Check received	$200		$200
CD		$50	$150
Book		$25	$125
Closing balance	$200	$75	$125

If these numbers were stored in variables, you could enter both the revenue and expenditure as positive values and only make the number negative when you want to calculate how much is left. You could do this by simply placing a minus sign (–) in front of the variable name.

To output the amount you had spent as a negative value, you could write the following:

```
int expenditure = 75;
printf("Your balance has changed by %d.", -expenditure);
```

This would result in the following output:

```
Your balance has changed by -75.
```

The minus sign will remind you that you've spent this money rather than gained it. Note that the expression `-expenditure` doesn't change the value stored in `expenditure`—it's still 75. The value of the *expression* is –75.

The unary minus operator in the expression `-expenditure` specifies an action, the result of which is the value of `expenditure` with its sign inverted: negative becomes positive and positive becomes negative. Instructions must be executed in your program to evaluate this. This is subtly different from when you use the minus operator when you write a negative number such as –75 or –1.25. In this case, the minus doesn't result in an action and no instructions need to be executed when your program is running. It simply instructs the compiler to create the appropriate negative constant in your program.

Variables and Memory

So far you've only looked at integer variables without considering how much space they take up in memory. Each time you declare a variable, the computer allocates a space in memory big enough to store that particular type of variable. Every variable of a particular type will always occupy the same amount of memory—the same number of bytes—but different types of variables require different amounts of memory to be allocated.

■ **Note** The amount of memory occupied by variables of a given type will always be the same on a particular machine. However, in some instances a variable of a given type on one computer may occupy more memory than it does on another. This is because the C language specification leaves it up to the compiler writer to decide how much memory a variable of a particular type will occupy. This allows the compiler writer to choose the size of a variable to suit the hardware architecture of the computer.

You saw at the beginning of this chapter how your computer's memory is organized into bytes. Each variable will occupy some number of bytes in memory, so how many bytes are needed to store an integer? Well, 1 byte can store an integer value from 128 to +127. This would be enough for the integer values that you've seen so far, but what if you want to store a count of the average number of stitches in a pair of knee-length socks? One byte wouldn't be anywhere near enough. Consequently, not only do you have variables of different types in C that store different types of numbers, one of which happens to be integers, you also have several varieties of integer variables to provide for different ranges of integers to be stored.

As I describe each type of variable in the following sections, I include a table containing the range of values that can be stored and the memory the variable will occupy. I summarize all these in a complete table of all the variable types in the "Summary" section of this chapter.

Integer Variable Types

You have five basic flavors of variables that you can declare that can store signed integer values (I'll get to unsigned integer values in the next section). Each type is specified by a different keyword or combination of keywords, as shown in Table 2-3.

Table 2-3. *Type Names for Integer Variable Types*

Type Name	Number of Bytes	Range of Values
`signed char`	1	128 to +127
`short int`	2	32,768 to +32,767
`int`	4	2,147,438,648 to +2,147,438,647
`long int`	4	2,147,438,648 to +2,147,438,647
`long long int`	8	9,223,372,036,854,775,808 to +9,223,372,036,854,775,807

The type names `short`, `long`, and `long long` can be used as abbreviations for the type names `short int`, `long int`, and `long long int`, and these types are almost always written in their abbreviated forms. Table 2-3 reflects the typical size of each type of integer variable, although the amount of memory occupied by variables of these types depends on the particular compiler you're using.

■ **Note** The only specific requirement imposed by the international standard for C on the integer types is that each type in the table won't occupy less memory than the type that precedes it. Type `unsigned char` occupies the same memory as type `char`, which has sufficient memory to store any character in the execution set for the language implementation; this is typically 1 byte but could be more. Outside of these constraints, the compiler-writer has complete freedom to make the best use of the hardware arithmetic capabilities of the machine on which the compiler is executing.

Unsigned Integer Types

For each of the types that store signed integers, there is a corresponding type that stores unsigned integers that occupy the same amount of memory as the unsigned type. Each unsigned type name is essentially the signed type name prefixed with the keyword `unsigned`. Table 2-4 shows the unsigned integer types that you can use.

Table 2-4. *Type Names for Unsigned Integer Types*

Type Name	Number of Bytes	Range of Values
`unsigned char`	1	0 to 255
`unsigned short int` or `unsigned short`	2	0 to 65,535
`unsigned int`	4	0 to 4,294,967,295

Table 2-4. *Type Names for Unsigned Integer Types*

Type Name	Number of Bytes	Range of Values
`unsigned long int` or `unsigned long`	4	0 to 4,294,967,295
`unsigned long long int` or `unsigned long long`	8	0 to +18,446,744,073,709,551,615

You use unsigned integer types when you are dealing with values that cannot be negative—the number of players in a football team for example, or the number of pebbles on a beach. With a given number of bits, the number of different values that can be represented is fixed. A 32-bit variable can represent any of 4,294,967,296 different values. Thus, using an unsigned type doesn't provide more values than the corresponding signed type, but it does allow numbers to be represented that are twice as large.

Using Integer Types

Most of the time variables of type `int` or `long` should suffice for your needs, with occasional requirements for unsigned `int` or unsigned `long`. Here are some examples of declarations of these types:

```
unsigned int count = 10;
unsigned long inchesPerMile = 63360UL;
int balance = -500;
```

Notice the `L` at the end of the value for the variable of type `long`. This identifies the constant as type `long` rather than type `int`; constants of type `int` have no suffix. Similarly the constant of type `unsigned long` has `UL` appended to it to identify it as that type. I come back to suffixes in the section "Specifying Integer Constants" later in this chapter.

Variables of type `int` should have the size most suited to the computer on which the code is executing. For example, consider the following statement:

```
int cookies = 0;
```

This statement will typically declare a variable to store integers that will occupy 4 bytes, but it could be 2 bytes with another compiler. This variation may seem a little strange, but the `int` type is intended to correspond to the size of integer that the computer has been designed to deal with most efficiently, and this can vary not only between different types of machines, but also with the same machine architecture, as the chip technology evolves over time. Ultimately, it's the compiler that determines what you get. Although at one time many C compilers for the PC created `int` variables as 2 bytes, with more recent C compilers on a PC, variables of type `int` occupy 4 bytes. This is because all modern processors move data around at least 4 bytes at a time. If your compiler is of an older vintage, it may still use 2 bytes for type `int`, even though 4 bytes would now be better on the hardware you're using.

■ **Note** The sizes of all these types are compiler-dependent. The international standard for the C language requires only that the size of `short` variables should be less than or equal to the size of type `int`, which in turn should be less than or equal to the size of type `long`.

If you use type `short`, you'll probably get 2-byte variables. The previous declaration could have been written as follows:

```
short cookies = 0;
```

Because the keyword `short` is actually an abbreviation for `short int`, you could write this as follows:

```
short int cookies = 0;
```

This is exactly the same as the previous statement. When you write just `short` in a variable declaration, the `int` is implied. Most people prefer to use this form—it's perfectly clear and it saves a bit of typing.

■ **Note** Even though type `short` and type `int` may occupy the same amount of memory on some machines, they're still different types.

If you need integers with a bigger range—to store the average number of hamburgers sold in one day, for instance—you can use the keyword `long`:

```
long Big_Number;
```

Type `long` defines an integer variable with a length of 4 bytes, which provides for a range of values from 2,147,438,648 to +2,147,438,647. As noted earlier, you can write `long int` if you wish instead of `long`, because it amounts to the same thing.

Specifying Integer Constants

Because you can have different kinds of integer variables, you might expect to have different kinds of integer constants, and you do. If you just write the integer value `100` for example, this will be of type `int`. If you want to make sure it is type `long`, you must append an uppercase or lowercase letter `L` to the numeric value. So the integer `100` as a `long` value is written as `100L`. Although it's perfectly legal, a lowercase letter `l` is best avoided because it's easily confused with the digit 1.

To declare and initialize the variable `Big_Number`, you could write this:

```
long Big_Number = 1287600L;
```

An integer constant will also be type `long` by default if it's outside the range of type `int`. Thus, if your compiler implementation uses 2 bytes to store type `int` values, the values `1000000` and `33000` will be of type `long` by default, because they won't fit into 2 bytes.

You write negative integer constants with a minus sign, for example:

```
int decrease = -4;
long  below_sea_level = -100000L;
```

You specify integer constants to be of type `long long` by appending two `L`s:

```
long long really_big_number = 123456789LL;
```

As you saw earlier, to specify a constant to be of an unsigned type you append a `U`, as in this example:

```
unsigned int count = 100U;
unsigned long value = 999999999UL;
```

You can also write integer values in hexadecimal form—that is, to base 16. The digits in a hexadecimal number are the equivalent of decimal values 0 to 15, and they're represented by `0` through `9` and `A` though `F` (or `a` through `f`). Because there needs to be a way to distinguish between 99_{10} and 99_{16}, hexadecimal numbers are written with the prefix `0x` or `0X`. You would therefore write 99_{16} in your program as `0x99` or as `0X99`.

Hexadecimal constants are most often used to specify bit patterns, because each hexadecimal digit corresponds to 4 binary bits. The bitwise operators that you'll see in Chapter 3 are usually used with hexadecimal constants that define masks. If you're unfamiliar with hexadecimal numbers, you can find a detailed discussion of them in Appendix A.

Note An integer constant that starts with a zero, such as 014 for example, will be interpreted by your compiler as an octal number—a number to base 8. Thus 014 is the octal equivalent of the decimal value 12. If it is meant to be the decimal value 14 it will be wrong, so don't put a leading zero in your integers unless you really mean to specify an octal value.

Floating-Point Values

Floating-point variables are used to store floating-point numbers. Floating-point numbers hold values that are written with a decimal point, so you can represent fractional as well as integral values. The following are examples of floating-point values:

```
1.6    0.00008    7655.899
```

Because of the way floating-point numbers are represented, they hold only a fixed number of decimal digits; however, they can represent a very wide range of values—much wider than integer types. Floating-point numbers are often expressed as a decimal value multiplied by some power of 10. For example, each of the previous examples of floating-point numbers could be expressed as shown in Table 2-5.

Table 2-5. *Expressing Floating-Point Numbers*

Value	With an Exponent	Can Also Be Written in C As
1.6	0.16×10^{1}	0.16E1
0.00008	0.8×10^{-4}	0.8E-4
7655.899	0.7655899×10^{4}	0.7655899E4

The center column shows how the numbers in the left column could be represented with an exponent. This isn't how you write them in C; it's just an alternative way of representing the same value designed to link to the right column. The right column shows how the representation in the center column would be expressed in C. The `E` in each of the numbers is for exponent, and you could equally well use a lowercase `e`. Of course, you can write each of these numbers in your program without an exponent, just as they appear in the left column, but for very large or very small numbers, the exponent form is very useful. I'm sure you would rather write `0.5E-15` than `0.0000000000000005`, wouldn't you?

Floating-Point Variables

There are three different types of floating-point variables, as shown in Table 2-6.

Table 2-6. *Floating-Point Variable Types*

Keyword	Number of Bytes	Range of Values
`float`	4	±3.4E38 (6 decimal digits precision)
`double`	8	±1.7E308 (15 decimal digits precision)
`long double`	12	±1.19E4932 (18 decimal digits precision)

These are typical values for the number of bytes occupied and the ranges of values that are supported. Like the integer types, the memory occupied and the range of values are dependent on the machine and the compiler. The type `long double` is sometimes exactly the same as type `double` with some compilers. Note that the number of decimal digits of precision is only an approximation because floating-point values will be stored internally in binary form, and a decimal floating-point value does not always have an exact representation in binary.

A floating-point variable is declared in a similar way to an integer variable. You just use the keyword for the floating-point type that you want to use:

```
float Radius;
double Biggest;
```

If you need to store numbers with up to seven digits of accuracy (a range of 10^{-38} to 10^{+38}), you should use variables of type `float`. Values of type `float` are known as **single precision** floating-point numbers**.** This type will occupy 4 bytes in memory, as you can see from the table. Using variables of type `double` will allow you to store **double precision** floating-point values. Each variable of type `double` will occupy 8 bytes in memory and give you 15-digit precision with a range of 10^{-308} to 10^{+308}. Variables of type `double` suffice for the majority of requirements, but some specialized applications require even more accuracy and range. The `long double` type provides the exceptional range and precision shown in the table.

To write a constant of type `float`, you append an `f` to the number to distinguish it from type `double`. You could initialize the last two variables with these statements:

```
float Radius = 2.5f;
double Biggest = 123E30;
```

The variable `Radius` has the initial value 2.5, and the variable `Biggest` is initialized to the number that corresponds to 123 followed by 30 zeroes. Any number that you write containing a decimal point is of type `double` unless you append the `F` to make it type `float`. When you specify an exponent value with `E` or `e`, the constant need not contain a decimal point. For instance, `1E3f` is of type `float` and `3E8` is of type `double`.

To specify a `long double` constant, you need to append an uppercase or lowercase letter `L` to the number as in the following example:

```
long double huge = 1234567.89123L;
```

Division Using Floating-Point Values

As you've seen, division operations with integer operands always produce an integer result. Unless the left operand of a division is an exact multiple of the right operand, the result will be inherently inaccurate. Of course, the way integer division works is an advantage if you're distributing cookies to children, but it isn't particularly useful when you want to cut a 10-foot plank into four equal pieces. This is a job for floating-point values.

Division operations with floating-point values will give you an exact result—at least, a result that is as exact as it can be with a fixed number of digits of precision. The next example illustrates how division operations work with variables of type `float`.

TRY IT OUT: DIVISION WITH VALUES OF TYPE FLOAT

Here's a simple example that just divides one floating-point value by another and displays the result:

```
/* Program 2.7 Division with float values */
#include <stdio.h>

int main(void)
{
  float plank_length = 10.0f;    /* In feet                  */
  float piece_count = 4.0f;      /* Number of equal pieces    */
  float piece_length = 0.0f;     /* Length of a piece in feet */

  piece_length = plank_length/piece_count;
  printf("A plank %f feet long can be cut into %f pieces %f feet long.",
                                      plank_length, piece_count, piece_length);
  return 0;
}
```

This program should produce the following output:

```
A plank 10.000000 feet long can be cut into 4.000000 pieces 2.500000 feet long.
```

How It Works

You shouldn't have any trouble understanding how you chop the plank into equal pieces. Note that you've used a new format specifier for values of type `float` in the `printf()` statement:

```
  printf("A plank %f feet long can be cut into %f pieces %f feet long.",
                                      plank_length, piece_count, piece_length);
```

You use the format specifier `%f` to display floating-point values. In general, the format specifier that you use must correspond to the type of value that you're outputting. If you output a value of type `float` with the specifier `%d` that's intended for use with integer values, you'll get rubbish. This is because the `float` value will be interpreted as an integer, which it isn't. Similarly, if you use `%f` with a value of an integer type, you'll also get rubbish as output.

Controlling the Number of Decimal Places

In the last example, you got a lot of decimal places in the output that you really didn't need. You may be good with a rule and a saw, but you aren't going to be able to cut the plank with a length of 2.500000 feet rather than 2.500001 feet. You can specify the number of places that you want to see after the decimal point in the format specifier. To obtain the output to two decimal places, you would write the format specifier as `%.2f`. To get three decimal places, you would write `%.3f`.

You can change the `printf()` statement in the last example so that it will produce more suitable output:

```
printf("A plank %.2f feet long can be cut into %.0f pieces %.2f feet long.",
                                     plank_length, piece_count, piece_length);
```

The first format specification corresponds to the `plank_length` variable and will produce output with two decimal places. The second specification will produce no decimal places—this makes sense here because the `piece_count` value is a whole number. The last specification is the same as the first. Thus, if you run the example with this version of the last statement, the output will be the following:

```
A plank 10.00 feet long can be cut into 4 pieces 2.50 feet long.
```

This is much more appropriate and looks a lot better.

Controlling the Output Field Width

The field width for the output, which is the total number of characters used for the value including spaces, has been determined by default. The `printf()` function works out how many character positions will be required for a value, given the number of decimal places you specify and uses that as the field width. However, you may want to decide the field width yourself. This will be the case if you want to output a column of values so they line up. If you let the `printf()` function work out the field width, you're likely to get a ragged column of output. A more general form of the format specifier for floating-point values can be written like this:

```
%[width][.precision][modifier]f
```

The square brackets here aren't part of the specification. They enclose bits of the specification that are optional, so you can omit the `width` or the `.precision` or the `modifier` or any combination of these. The `width` value is an integer specifying the total number of characters in the output: the field width. The `precision` value is an integer specifying the number of decimal places that are to appear after the decimal point. The `modifier` part is `L` when the value you are outputting is type `long double`, otherwise you omit it.

You could rewrite the `printf()` call in the last example to specify the field width as well as the number of digits you want after the decimal point, as in the following example:

```
printf("A %8.2f plank foot can be cut into %5.0f pieces %6.2f feet long.",
                                     plank_length, piece_count, piece_length);
```

I changed the text a little to get it to fit across the page here. The first value now will have a field width of 8 and 2 decimal places after the decimal point. The second value, which is the count of the number of pieces, will have a field width of 5 characters and no decimal places. The third value will be presented in a field width of 6 with 2 decimal places.

When you specify the field width, the value will be right-aligned by default. If you want the value to be left-aligned in the field, just put a minus sign following the `%`. For instance, the specification `%-10.4f` will output a floating-point value left-aligned in a field width of 10 characters with 4 digits following the decimal point.

Note that you can specify a field width and the alignment in the field with a specification for outputting an integer value. For example, `%-15d` specifies an integer value will be presented left-aligned in a field width of 15 characters.

There's more to format specifiers than I've introduced here, and you'll learn more about them later. Try out some variations using the previous example. In particular, see what happens when the field width is too small for the value.

More Complicated Expressions

You know that arithmetic can get a lot more complicated than just dividing a couple of numbers. In fact, if that is all you are trying to do, you may as well use paper and pencil. Now that you have the tools of addition, subtraction, multiplication, and division at your disposal, you can start to do some really heavy calculations.

For these more complicated calculations, you'll need more control over the sequence of operations when an expression is evaluated. Parentheses provide you with this capability. They can also help to make expressions clearer when they're getting intricate.

You can use parentheses in arithmetic expressions, and they work much as you'd expect. Subexpressions contained within parentheses are evaluated in sequence from the innermost pair of parentheses to the outermost, with the normal rules that you're used to for operator precedence, where multiplication and division happen before addition or subtraction. Therefore, the expression 2 * (3 + 3 * (5 + 4)) evaluates to 60. You start with the expression 5 + 4, which produces 9. Then you multiply that by 3, which gives 27. Then you add 3 to that total (giving 30) and multiply the whole lot by 2.

You can insert spaces to separate operands from operators to make your arithmetic statements more readable, or you can leave them out when you need to make the code more compact. Either way, the compiler doesn't mind, as it will ignore the spaces. If you're not quite sure of how an expression will be evaluated according to the precedence rules, you can always put in some parentheses to make sure it produces the result you want.

TRY IT OUT: ARITHMETIC IN ACTION

This time you'll have a go at calculating the circumference and area of a circular table from an input value for its diameter radius. You may remember from elementary math the equations to calculate the area and circumference of a circle using π or pi (circumference = 2πr and area = πr2, where *r* is the radius). If you don't, don't worry. This isn't a math book, so just look at how the program works.

```
/* Program 2.8 calculations on a table */
#include <stdio.h>

int main(void)
{
  float radius = 0.0f;           /* The radius of the table       */
  float diameter = 0.0f;         /* The diameter of the table     */
  float circumference = 0.0f;    /* The circumference of the table */
  float area = 0.0f;             /* The area of a circle          */
  float Pi = 3.14159265f;
  printf("Input the diameter of the table:");
  scanf("%f", &diameter);         /* Read the diameter from the keyboard */
  radius = diameter/2.0f;         /* Calculate the radius                */
  circumference = 2.0f*Pi*radius; /* Calculate the circumference         */
  area = Pi*radius*radius;        /* Calculate the area                  */
  printf("\nThe circumference is %.2f", circumference);
  printf("\nThe area is %.2f\n", area);
  return 0;
}
```

Here's some typical output from this example:

```
Input the diameter of the table: 6

The circumference is 18.85
The area is 28.27
```

How It Works

Up to the first `printf()`, the program looks much the same as those you've seen before:

```
  float radius = 0.0f;                /* The radius of the table        */
  float diameter = 0.0f;              /* The diameter of the table      */
  float circumference = 0.0f;         /* The circumference of the table */
  float area = 0.0f;                  /* The area of a circle           */
  float Pi = 3.14159265f;
```

You declare and initialize five variables, where `Pi` has its usual value. Note how all the initial values have an `f` at the end because you're initializing values of type `float`. Without the `f` the values would be of type `double`. They would still work here, but you would be introducing some unnecessary conversion that the compiler would have to arrange, from type `double` to type `float`.

The next statement outputs a prompt for input from the keyboard:

```
  printf("Input the diameter of the table:");
```

The next statement deals with reading the value for the diameter of the table. You use a new standard library function, the `scanf()` function, to do this:

```
  scanf("%f", &diameter);           /* Read the diameter from the keyboard */
```

The `scanf()` function is another function that requires the `<stdio.h>` header file to be included. This function handles input from the keyboard. In effect it takes what you enter through the keyboard and interprets it as specified by the first argument, which is a control string between double quotes. In this case the control string is "%f" because you're reading a value of type `float`. It stores the result in the variable specified by the second argument, `diameter` in this instance. The first argument is a control string similar to what you've used with the `printf()` function, except that here it controls input rather than output. You'll learn more about the `scanf()` function in Chapter 10 and, for reference, Appendix D summarizes the control strings you can use with it.

You've undoubtedly noticed something new here: the & preceding the variable name `diameter`. This is called the `address of` operator, and it's needed to allow the `scanf()` function to store the value that is read in your variable, `diameter`. The reason for this is bound up with the way argument values are passed to a function. For the moment, I won't go into a more detailed explanation of this; you'll see more on this in Chapter 8. The only thing to remember is to use the `address of` operator (the & sign) before a variable when you're using the `scanf()` function, and not to use it when you use the `printf()` function.

Within the control string for the `scanf()` function, the `%` character identifies the start of a format specification for an item of data. The `f` that follows the `%` indicates that the input is a floating-point value. In general there can be several format specifications within the control string, in which case these determine the type of data for each of the subsequent arguments to the function in sequence. You'll see a lot more on how `scanf()` works later in the book, but for now the basic set of format specifiers you can use for reading data of various types are shown in the following table.

Format Specifiers for Reading Data

Action	**Required Control String**
To read a value of type `short`	`%hd`
To read a value of type `int`	`%d`
To read a value of type `long`	`%ld`
To read a value of type `float`	`%f` or `%e`
To read a value of type `double`	`%lf` or `%le`

In the `%ld` and `%lf` format specifiers, `l` is a lowercase letter `L`. Don't forget, you must *always* prefix the name of the variable that's receiving the input value with `&`. Also, if you use the wrong format specifier—if you read a value into a variable of type `float` with `%d`, for instance—the data value in your variable won't be correct, but you'll get no indication that a junk value has been stored.

Next, you have two statements that calculate the results you're interested in:

```
  radius = diameter/2.0f;               /* Calculate the radius        */
  circumference = 2.0f*Pi*radius;       /* Calculate the circumference */
  area = Pi*radius*radius;              /* Calculate the area          */
```

The first statement calculates the radius as half of the value of the diameter that was entered. The second statement computes the circumference of the table, using the value that was calculated for the radius. The third statement calculates the area. Note that if you forget the `f` in `2.0f`, you'll probably get a warning message from your compiler. This is because without the `f`, the constant is of type `double`, and you would be mixing different types in the same expression. You'll see more about this later.

The next two statements output the values you've calculated:

```
  printf("\nThe circumference is %.2f", circumference);
  printf("\nThe area is %.2f\n", area);
```

These two `printf()` statements output the values of the variables `circumference` and `area` using the format specifier `%.2f`. As you've already seen, in both statements the format control string contains text to be displayed, as well as a format specifier for the variable to be output. The format specification outputs the values with two decimal places after the point. The default field width will be sufficient in each case to accommodate the value that is to be displayed.

Of course, you can run this program and enter whatever values you want for the diameter. You could experiment with different forms of floating-point input here, and you could try entering something like `1E1f`, for example.

Defining Constants

Although Pi is defined as a variable in the previous example, it's really a constant value that you don't want to change. The value of π is always a fixed number with an unlimited number of decimal digits. The only question is how many digits of precision you want to use in its specification. It would be nice to make sure its value stayed fixed in a program so it couldn't be changed by mistake.

There are a couple of ways in which you can approach this. The first is to define Pi as a symbol that's to be replaced in the program by its value during compilation. In this case, Pi isn't a variable at all, but more a sort of alias for the value it represents. Let's try that out.

TRY IT OUT: DEFINING A CONSTANT

Let's look at specifying PI as an alias for its value:

```
/* Program 2.9 More round tables */
#include <stdio.h>
#define PI   3.14159f             /* Definition of the symbol PI */

int main(void)
{
  float radius = 0.0f;
  float diameter = 0.0f;
  float circumference = 0.0f;
  float area = 0.0f;

  printf("Input the diameter of a table:");
  scanf("%f", &diameter);
  radius = diameter/2.0f;
  circumference = 2.0f*PI*radius;
  area = PI*radius*radius;
  printf("\nThe circumference is %.2f", circumference);
  printf("\nThe area is %.2f", area);
  return 0;
}
```

This produces exactly the same output as the previous example.

How It Works

After the comment and the #include directive for the header file, there is a preprocessing directive:

```
#definePI 3.14159f                  /* Definition of the symbol PI */
```

You've now defined PI as a symbol that is to be replaced in the code by 3.14159f. You've used PI rather than Pi, as it's a common convention in C to write identifiers that appear in a #define statement in capital letters. Wherever you reference PI within an expression in the program, the preprocessor will substitute the value you've specified for it in the #define directive. All the substitutions will be made before compiling the program. When the program is ready to be compiled, it will no longer contain references to PI, as all occurrences will have been replaced by the sequence of characters that you've specified in the #define directive. This all happens internally while your program is processed by the compiler. Your source program will not be changed; it will still contain the symbol PI.

The second possibility is to define `Pi` as a variable, but to tell the compiler that its value is fixed and must not be changed. You can fix the value of any variable by prefixing the type name with the keyword `const` when you declare the variable, for example:

```
const float Pi = 3.14159f;        /* Defines the value of Pi as fixed */
```

The advantage of defining `Pi` in this way is that you are now defining it as a constant numerical value. In the previous example `PI` was just a sequence of characters that replaced all occurrences of `PI` in your code.

Adding the keyword `const` in the declaration for `Pi` will cause the compiler to check that the code doesn't attempt to change its value. Any code that does so will be flagged as an error and the compilation will fail. Let's see a working example of this.

TRY IT OUT: DEFINING A VARIABLE WITH A FIXED VALUE

Try using a constant in a variation of the previous example but with the code shortened a little:

```
/* Program 2.10 Round tables again but shorter */
#include <stdio.h>

int main(void)
{
  float diameter = 0.0f;           /* The diameter of a table          */
  float radius = 0.0f;             /* The radius of a table            */
  const float Pi = 3.14159f;       /* Defines the value of Pi as fixed */

  printf("Input the diameter of the table:");
  scanf("%f", &diameter);
  radius = diameter/2.0f;
  printf("\nThe circumference is %.2f", 2.0f*Pi*radius);
  printf("\nThe area is %.2f", Pi*radius*radius);
  return 0;
}
```

How It Works

Following the declaration for the variable `radius` is this statement:

```
  const float Pi = 3.14159f;        /* Defines the value of Pi as fixed */
```

This declares the variable `Pi` and defines a value for it; `Pi` is still a variable here, but the initial value you've given it can't be changed. The `const` modifier achieves this effect. It can be applied to any statement declaring a variable of any type to fix the value of that variable. Of course, the value must appear in the declaration in the same way as shown here: following an = sign after the variable name. The compiler will check your code for attempts to change variables that you've declared as `const`, and if it discovers that you've attempted to change a `const` variable it will complain. There are ways to trick the compiler to change `const` variables, but this defeats the whole point of using `const` in the first place.

The next two statements produce the output from the program:

```
  printf("\nThe circumference is %.2f", 2.0f*Pi*radius);
  printf("\nThe area is %.2f", Pi*radius*radius);
```

In this example, you've done away with the variables storing the circumference and area of the circle. The expressions for these now appear as arguments in the `printf()` statements where they're evaluated, and their values are passed directly to the function.

As you've seen before, the value that you pass to a function can be the result of evaluating an expression rather than the value of a particular variable. The compiler will create a temporary variable to hold the value and that will be passed to the function. The temporary variable is subsequently discarded. This is fine, as long as you don't want to use these values elsewhere.

Knowing Your Limitations

Of course, it may be important to be able to determine within a program exactly what the limits are on the values that can be stored by a given integer type. The header file `<limits.h>` defines symbols representing values for the limits for each type. Table 2-7 shows the symbols names corresponding to the limits for each signed type.

Table 2-7. *Symbols Representing Range Limits for Integer Types*

Type	Lower Limit	Upper Limit
`char`	`CHAR_MIN`	`CHAR_MAX`
`short`	`SHRT_MIN`	`SHRT_MAX`
`int`	`INT_MIN`	`INT_MAX`
`long`	`LONG_MIN`	`LONG_MAX`
`long long`	`LLONG_MIN`	`LLONG_MAX`

The lower limits for the unsigned integer types are all 0 so there are no symbols for these. The symbols corresponding to the upper limits for the unsigned integer types are `UCHAR_MAX`, `USHRT_MAX`, `UINT_MAX`, `ULONG_MAX`, and `ULLONG_MAX`.

To be able to use any of these symbols in a program you must have an `#include` directive for the `<limits.h>` header file in the source file:

```
#include <limits.h>
```

You could initialize a variable with the maximum possible value like this:

```
int number = INT_MAX;
```

This statement sets the value of `number` to be the maximum possible, whatever that may be for the compiler used to compile the code.

The `<float.h>` header file defines symbols that characterize floating-point values. Some of these are quite technical so I'll just mention those you are most likely to be interested in. The maximum and minimum positive values that can be represented by the three floating-point types are shown in Table 2-8.

You can also access the symbols `FLT_DIG`, `DBL_DIG`, and `LDBL_DIG` that indicate the number of decimal digits that can be represented by the binary mantissa of the corresponding types.

Let's explore in a working example how to access some of the symbols characterizing integers and floating-point values.

Table 2-8. *Symbols Representing Range Limits for Floating-Point Types*

Type	Lower Limit	Upper Limit
float	FLT_MIN	FLT_MAX
double	DBL_MIN	DBL_MAX
long double	LDBL_MIN	LDBL_MAX

TRY IT OUT: FINDING THE LIMITS

This program just outputs the values corresponding to the symbols defined in the header files:

```
/* Program 2.11 Finding the limits  */
#include <stdio.h>      /* For command line input and output  */
#include <limits.h>     /* For limits on integer types        */
#include <float.h>      /* For limits on floating-point types */

int main(void)
{
  printf("Variables of type char store values from %d to %d",
                                                  CHAR_MIN, CHAR_MAX);
  printf("\nVariables of type unsigned char store values from 0 to %u",
                                                           UCHAR_MAX);
  printf("\nVariables of type short store values from %d to %d",
                                                  SHRT_MIN, SHRT_MAX);
  printf("\nVariables of type unsigned short store values from 0 to %u",
                                                           USHRT_MAX);
  printf("\nVariables of type int store values from %d to %d", INT_MIN,
                                                             INT_MAX);
  printf("\nVariables of type unsigned int store values from 0 to %u",
                                                            UINT_MAX);
  printf("\nVariables of type long store values from %ld to %ld",
                                                  LONG_MIN, LONG_MAX);
  printf("\nVariables of type unsigned long store values from 0 to %lu",
                                                           ULONG_MAX);
  printf("\nVariables of type long long store values from %lld to %lld",
                                                LLONG_MIN, LLONG_MAX);
  printf("\nVariables of type unsigned long long store values from 0 to %llu",
                                                          ULLONG_MAX);

  printf("\n\nThe size of the smallest non-zero value of type float is %.3e",
                                                             FLT_MIN);
  printf("\nThe size of the largest value of type float is %.3e", FLT_MAX);
  printf("\nThe size of the smallest non-zero value of type double is %.3e",
                                                             DBL_MIN);
  printf("\nThe size of the largest value of type double is %.3e", DBL_MAX);
  printf("\nThe size of the smallest non-zero value ~CCC
 of type long double is %.3Le", LDBL_MIN);
  printf("\nThe size of the largest value of type long double is %.3Le\n",
                                                            LDBL_MAX);
  printf("\nVariables of type float provide %u decimal digits precision.",
                                                           FLT_DIG);
  printf("\nVariables of type double provide %u decimal digits precision.",
                                                           DBL_DIG);
  printf("\nVariables of type long double provide %u decimal digits precision.",
                                                          LDBL_DIG);
  return 0;
}
```

You'll get output somewhat similar to the following:

```
Variables of type char store values from -128 to 127
Variables of type unsigned char store values from 0 to 255
Variables of type short store values from -32768 to 32767
Variables of type unsigned short store values from 0 to 65535
Variables of type int store values from -2147483648 to 2147483647
Variables of type unsigned int store values from 0 to 4294967295
Variables of type long store values from -2147483648 to 2147483647
Variables of type unsigned long store values from 0 to 4294967295
Variables of type long long store values ~CCC
from -9223372036854775808 to 9223372036854775807
Variables of type unsigned long long store values from 0 to 18446744073709551615

The size of the smallest non-zero value of type float is 1.175e-038
The size of the largest value of type float is 3.403e+038
The size of the smallest non-zero value of type double is 2.225e-308
The size of the largest value of type double is 1.798e+308
The size of the smallest non-zero value of type long double is 3.362e-4932
The size of the largest value of type long double is 1.190e+4932

Variables of type float provide 6 decimal digits precision.
Variables of type double provide 15 decimal digits precision.
Variables of type long double provide 18 decimal digits precision.
```

How It Works

You output the values of symbols that are defined in the `<limits.h>` and `<float.h>` header files in a series of `printf()` function calls. Numbers in your computer are always limited in the range of values that can be stored, and the values of these symbols represent the boundaries for values of each numerical type. You have used the `%u` specifier to output the unsigned integer values. If you use `%d` for the maximum value of an unsigned type, values that have the leftmost bit (the sign bit for signed types) as 1 won't be interpreted correctly.

You use the `%e` specifier for the floating-point limits, which presents the values in exponential form. You also specify just three digits precision, as you don't need the full accuracy in the output. The `L` modifier is necessary when the value being displayed by the `printf()` function is type `long double`. Remember, this has to be a capital letter `L`; a small letter `l` won't do here. The `%f` specifier presents values without an exponent, so it's rather inconvenient for very large or very small values. If you try it in the example, you'll see what I mean.

Introducing the sizeof Operator

You can find out how many bytes are occupied by a given type by using the `sizeof` operator. Of course, `sizeof` is a keyword in C. The expression `sizeof(int)` will result in the number of bytes occupied by a variable of type `int`, and the value that results is an integer of type `size_t`. Type `size_t` is defined in the standard header file `<stddef.h>` (as well as possibly other header files such as `<stdio.h>`) and will correspond to one of the basic integer types. However, because the choice of type that corresponds to type `size_t` may differ between one C library and another, it's best to use variables of `size_t` to store the value produced by the `sizeof` operator, even when you know which basic type it corresponds to. Here's how you could store a value that results from applying the `sizeof` operator:

```
size_t size = sizeof(long long);
```

You can also apply the `sizeof` operator to an expression, in which case the result is the size of the value that results from evaluating the expression. In this context the expression would usually be just a variable of some kind.

The `sizeof` operator has uses other than just discovering the memory occupied by a value of a basic type, but for the moment let's just use it to find out how many bytes are occupied by each type.

TRY IT OUT: DISCOVERING THE NUMBER OF BYTES OCCUPIED BY A GIVEN TYPE

This program will output the number of bytes occupied by each numeric type:

```
/* Program 2.12 Finding the size of a type  */
#include <stdio.h>

int main(void)
{
  printf("\nVariables of type char occupy %d bytes", sizeof(char));
  printf("\nVariables of type short occupy %d bytes", sizeof(short));
  printf("\nVariables of type int occupy %d bytes", sizeof(int));
  printf("\nVariables of type long occupy %d bytes", sizeof(long));
  printf("\nVariables of type float occupy %d bytes", sizeof(float));
  printf("\nVariables of type double occupy %d bytes", sizeof(double));
  printf("\nVariables of type long double occupy %d bytes",
                                                sizeof(long double));
  return 0;
}
```

On my system I get the following output:

```
Variables of type char occupy 1 bytes
Variables of type short occupy 2 bytes
Variables of type int occupy 4 bytes
Variables of type long occupy 4 bytes
Variables of type float occupy 4 bytes
Variables of type double occupy 8 bytes
Variables of type long double occupy 12 bytes
```

How It Works

Because the `sizeof` operator results in an integer value, you can output it using the `%d` specifier. Note that you can also obtain the number of bytes occupied by a variable, `var_name`, with the expression `sizeof var_name`. Obviously, the space between the `sizeof` keyword and the variable name in the expression is essential.

Now you know the range limits and the number of bytes occupied by each numeric type with your compiler.

Choosing the Correct Type for the Job

You have to be careful to select the type of variable that you're using in your calculations so that it accommodates the range of values that you expect. If you use the wrong type, you may find that errors creep into your programs that can be hard to detect. This is best shown with an example.

TRY IT OUT: THE RIGHT TYPES OF VARIABLES

Here's an example of how things can go horribly wrong if you choose an unsuitable type for your variables:

```
/* Program 2.13 Choosing the correct type for the job  1*/
#include <stdio.h>

int main(void)
{
  const float Revenue_Per_150 = 4.5f;
  short JanSold = 23500;                      /* Stock sold in January  */
  short FebSold = 19300;                      /* Stock sold in February */
  short MarSold = 21600;                      /* Stock sold in March    */
  float  RevQuarter = 0.0f;                   /* Sales for the quarter  */

  short QuarterSold = JanSold+FebSold+MarSold; /* Calculate quarterly total */

  /* Output monthly sales and total for the quarter */
  printf("\nStock sold in\n Jan: %d\n Feb: %d\n Mar: %d",
                                                JanSold,FebSold,MarSold);
  printf("\nTotal stock sold in first quarter: %d",QuarterSold);

  /* Calculate the total revenue for the quarter and output it */
   RevQuarter = QuarterSold/150*Revenue_Per_150;
  printf("\nSales revenue this quarter is:$%.2f\n",RevQuarter);
  return 0;
}
```

These are fairly simple calculations, and you can see that the total stock sold in the quarter should be 64400. This is just the sum of each of the monthly totals, but if you run the program, the output you get is this:

```
Stock sold in
 Jan: 23500
 Feb: 19300
 Mar: 21600
Total stock sold in first quarter: -1136
Sales revenue this quarter is:$-31.50
```

Obviously all is not right here. It doesn't take a genius or an accountant to tell you that adding three big, positive numbers together shouldn't give a negative result!

How It Works

First you define a constant that will be used in the calculation:

```
  const Revenue_Per_150 = 4.5f;
```

This defines the revenue obtained for every 150 items sold. There's nothing wrong with that.

Next, you declare four variables and assign initial values to them:

```
  short JanSold = 23500;                        /* Stock sold in January  */
  short FebSold = 19300;                        /* Stock sold in February */
  short MarSold = 21600;                        /* Stock sold in March    */
  float  RevQuarter = 0.0f;                     /* Sales for the quarter  */
```

The first three variables are of type `short`, which is quite adequate to store the initial value. The `RevQuarter` variable is of type `float` because you want two decimal places for the quarterly revenue.

The next statement declares the variable `QuarterSold` and stores the sum of the sales for each of the months:

```
  short QuarterSold = JanSold+FebSold+MarSold; /* Calculate quarterly total */
```

Note that you're initializing this variable with the result of an expression. This is only possible because the values of these variables are known to the compiler, so this represents what is known as a **constant expression**. If any of the values in the expression were determined during execution of the program—from a calculation involving a value that was read in, for instance—this wouldn't compile. The compiler can only use initial values that are explicit or are produced by an expression that the compiler can evaluate.

In fact, the cause of the erroneous results is in the declaration of the `QuarterSold` variable. You've declared it to be of type `short` and given it the initial value of the sum of the three monthly figures. You know that their sum is 64400 and that the program outputs a negative number. The error must therefore be in this statement.

The problem arises because you've tried to store a number that's too large for type `short`. If you remember, the maximum value that a `short` variable can hold is 32,767. The computer can't interpret the value of `QuarterSold` correctly and happens to give a negative result. The solution to your problem is to use a variable of type `long` that will allow you to store much larger numbers.

Solving the Problem

Try changing the program and running it again. You need to change only two lines in the body of the function `main()`. The new and improved program is as follows:

```
/* Program 2.14 Choosing the correct type for the job  2 */
#include <stdio.h>

int main(void)
{
  const float Revenue_Per_150 = 4.5f;
  short JanSold =23500;              /* Stock sold in January  */
  short FebSold =19300;              /* Stock sold in February */
  short MarSold =21600;              /* Stock sold in March    */
  float  RevQuarter = 0.0f;          /* Sales for the quarter */

  long QuarterSold = JanSold+FebSold+MarSold; /* Calculate quarterly total */

  /* Output monthly sales and total for the quarter */
  printf("Stock sold in\n Jan: %d\n Feb: %d\n Mar: %d\n",
                                            JanSold,FebSold,MarSold);
  printf("Total stock sold in first quarter: %ld\n",QuarterSold);

  /* Calculate the total revenue for the quarter and output it */
  RevQuarter = QuarterSold/150*Revenue_Per_150;
  printf("Sales revenue this quarter is:$%.2f\n",RevQuarter);
  return 0;
}
```

When you run this program, the output is more satisfactory:

```
Stock sold in
 Jan: 23500
 Feb: 19300
 Mar: 21600
Total stock sold in first quarter: 64400
Sales revenue this quarter is :$1930.50
```

The stock sold in the quarter is correct, and you have a reasonable result for revenue. Notice that you use `%ld` to output the total stock sold. This is to tell the compiler that it is to use a `long` conversion for the output of this value. Just to check the program, calculate the result of the revenue yourself with a calculator.

The result you should get is, in fact, $1,932. Somewhere you've lost a dollar and a half. Not such a great amount, but try saying that to an accountant. You need to find the lost $1.50. Consider what's happening when you calculate the value for revenue in the program.

```
RevQuarter = QuarterSold/150*Revenue_Per_150;
```

Here you're assigning a value to `RevQuarter`. The value is the result of the expression on the right of the = sign. The result of the expression will be calculated, step by step, according to the precedence rules you've already looked at in this chapter. Here you have quite a simple expression that's calculated from left to right, as division and multiplication have the same priority. Let's work through it:

- `QuarterSold/150` is calculated as 64400 / 150, which should produce the result 429.333.

This is where your problem arises. `QuarterSold` is an integer and so the computer truncates the result of the division to an integer, ignoring the .333. This means that when the next part of the calculation is evaluated, the result will be slightly off.

- `429*Revenue_Per_150` is calculated as 429 * 4.5 which is 1930.50.

You now know where the error has occurred, but what can you do about it? You could change all of your variables to floating-point types, but that would defeat the purpose of using integers in the first place. The numbers entered really are integers, so you'd like to store them as such. Is there an easy solution to this? In this case there is. You can rewrite the statement as follows:

```
RevQuarter = Revenue_Per_150*QuarterSold/150;
```

Now the multiplication will occur first; and because of the way arithmetic with mixed operands works, the result will be of type `float`. The compiler will automatically arrange for the integer operand to be converted to floating-point. When you then divide by 150, that operation will execute with `float` values too, with `150` being converted to `150f`. The net effect is that the result will now be correct.

However, there's more to it than that. Not only do you need to understand more about what happens with arithmetic between operands of different types, but also you need to understand how you can control conversions from one type to another. In C you have the ability to explicitly convert a value of one type to another type. This process is called **casting**.

Explicit Type Conversion

Let's look again at the original expression to calculate the quarterly revenue that you saw in Program 2.14 and see how you can control what goes on so that you end up with the correct result:

```
  RevQuarter = QuarterSold/150*Revenue_Per_150;
```

You know that if the result is to be correct, this statement has to be amended so that the expression is calculated in floating-point form. If you can convert the value of `QuarterSold` to type `float`, the expression will be evaluated as floating-point and your problem will be solved. To convert the value of a variable to another type, place the type that you want to cast the value to in parentheses in front of the variable. Thus, the statement to calculate the result correctly will be the following:

```
  RevQuarter = (float)QuarterSold/150.0f*Revenue_Per_150;
```

This is exactly what you require. You're using the right types of variables in the right places. You're also ensuring you don't use integer arithmetic when you want to keep the fractional part of the result of a division. An explicit conversion from one type to another is called a **cast**.

Automatic Conversion

Look at the output from the second version of the program again:

```
Sales revenue this quarter is :$1930.50
```

Even without the explicit cast in the expression, the result is in floating-point form, even though it is still wrong. This is because the compiler automatically converts one of the operands to be the same type as the other when it's dealing with an operation that involves values of different types.

Binary arithmetic operations (add, subtract, multiply, divide, and remainder) can only be executed by your computer when both operands are of the same type. When you use operands in a binary operation that are of different types, the compiler arranges for the value that is of a type with a more limited range to be converted to the type of the other. This is referred to as an **implicit conversion**. So referring back to the expression to calculate revenue

```
   QuarterSold / 150 * Revenue_Per_150
```

it evaluated as 64400 (`int`) / 150 (`int`), which equals 429 (`int`). Then 429 (`int` converted to `float`) is multiplied by 4.5 (`float`), giving 1930.5 (`float`).

An implicit conversion always applies when a binary operator involves operands of different types. With the first operation, the numbers are both of type `int`, so the result is of type `int`. With the second operation, the first value is type `int` and the second value is type `float`. Type `int` is more limited in its range than type `float`, so the value of type `int` is automatically cast to type `float`. Whenever there is a mixture of types in an arithmetic expression, your C compiler will use a set of specific rules to decide how the expression will be evaluated. Let's have a look at these rules now.

Rules for Implicit Conversions

The mechanism that determines which operand in a binary operation is to be changed to the type of the other is relatively simple. Broadly it works on the basis that the operand with the type that has the more restricted range of values will be converted to the type of the other operand, although in some instances both operands will be promoted.

To express accurately in words how this works is somewhat more complicated than the description in the previous paragraph, so you may want to ignore the fine detail that follows and maybe refer back to it if you need to. If you do want the full story, read on.

The compiler determines the implicit conversion to use by applying the following rules in sequence:

1. If one operand is of type `long double` the other operand will be converted to type `long double`.
2. Otherwise, if one operand is of type `double` the other operand will be converted to type `double`.
3. Otherwise, if one operand is of type `float` the other operand will be converted to type `float`.
4. Otherwise, if the operands are both of signed integer types, or both of unsigned integer types, the operand of the type of lower rank is converted to the type of the other operand.

   The unsigned integer types are ranked from low to high in the following sequence: `signed char`, `short`, `int`, `long`, `long long`.

   Each unsigned integer type has the same rank as the corresponding signed integer type, so type `unsigned int` has the same rank as `type int`, for example.
5. Otherwise, if the operand of the signed integer type has a rank that is less than or equal to the rank of the unsigned integer type, the signed integer operand is converted to the unsigned integer type.
6. Otherwise, if the range of values the signed integer type can represent includes the values that can be represented by the unsigned integer type, the unsigned operand is converted to the signed integer type.
7. Otherwise, both operands are converted to the unsigned integer type corresponding to the signed integer type.

Implicit Conversions in Assignment Statements

You can also cause an implicit conversion to be applied when the value of the expression on the right of the assignment operator is a different type to the variable on the left. In some circumstances this can cause values to be truncated so information is lost. For instance, if an assignment operation stores a value of type `float` or `double` to a variable of type `int` or `long`, the fractional part of the `float` or `double` will be lost, and just the integer part will be stored. The following code fragment illustrates this situation:

```
int number = 0;
float value = 2.5f;
number = value;
```

The value stored in `number` will be `2`. Because you've assigned the value of `decimal (2.5)` to the variable, `number`, which is of type `int`, the fractional part, `.5`, will be lost and only the `2` will be stored. Notice how I've used a specifier `f` at the end of `2.5f`.

An assignment statement that may lose information because an automatic conversion has to be applied will usually result in a warning from the compiler. However, the code will still compile, so there's a risk that your program may be doing things that will result in incorrect results. Generally, it's better to put explicit casts in your code wherever conversions that may result in information being lost are necessary.

Let's look at an example to see how the conversion rules in assignment operations work in practice. Look at the following code fragment:

```
double price = 10.0;          /* Product price per unit  */
long count = 5L;              /* Number of items         */
float ship_cost = 2.5F;       /* Shipping cost per order */
int discount = 15;            /* Discount as percentage  */
long double total_cost = (count*price + ship_cost)*((100L - discount)/100.0F);
```

This declares the four variables that you see and computes the total cost of an order from the values set for these variables. I chose the types primarily to demonstrate implicit conversions, and these types would not represent a sensible choice in normal circumstances. Let's see what happens in the last statement to produce the value for `total_cost`:

1. `count*price` is evaluated first and `count` will be implicitly converted to type `double` to allow the multiplication to take place and the result will be of type `double`. This results from the second rule.
2. Next `ship_cost` is added to the result of the previous operation and, to make this possible, the value of `ship_cost` is converted to the value of the previous result, type `double`. This conversion also results from the second rule.
3. Next, the expression `100L - discount` is evaluated, and to allow this to occur the value of `discount` will be converted to type `long`, the type of the other operand in the subtraction. This is a result of the fourth rule and the result will be type `long`.
4. Next, the result of the previous operation (of type `long`) is converted to type `float` to allow the division by `100.0F` (of type `float`) to take place. This is the result of applying the third rule, and the result is of type `float`.
5. The result of step 2 is divided by the result of step 4, and to make this possible the float value from the previous operation is converted to type double. This is a consequence of applying the third rule, and the result is of type double.
6. Finally, the previous result is stored in the variable `total_cost` as a result of the assignment operation. An assignment operation always causes the type of the right operand to be converted to that of the left when the operand types are different, regardless of the types of the operands, so the result of the previous operation is converted to type `long double`. No compiler warning will occur because all values of type `double` can be represented as type `long double`.

More Numeric Data Types

To complete the set of numeric data types, I'll now cover those that I haven't yet discussed. The first is one that I mentioned previously: type `char`. A variable of type `char` can store the code for a single character. Because it stores a character code, which is an integer, it's considered to be an integer type. Because it's an integer type, you can treat the value stored just like any other integer so you can use it in arithmetic calculations.

The Character Type

Values of type `char` occupy the least amount of memory of all the data types. They typically require just 1 byte. The integer that's stored in a variable of type `char` can be interpreted as a signed or unsigned value, depending on your compiler. As an unsigned type, the value stored in a variable of type `char` can range from 0 to 255. As a signed type, a variable of type `char` can store values from –128 to +127. Of course, both ranges correspond to the same set of bit patterns: from 0000 0000 to 1111 1111. With unsigned values, all eight bits are data bits, so 0000 0000 corresponds to 0, and 1111 1111 corresponds to 255. With unsigned values, the leftmost bit is a sign bit, so –128 is the binary value 1000 0000, 0 is 0000 0000, and 127 is 0111 1111. The value 1111 1111 as a signed binary value is the decimal value –1.

Thus, from the point of view of representing character codes, which are bit patterns, it doesn't matter whether type char is regarded as signed or unsigned. Where it *does* matter is when you perform arithmetic operations on values of type char.

A char variable can hold any single character, so you can specify the initial value for a variable of type char by a character constant. A **character constant** is a character written between single quotes. Here are some examples:

```
char letter = 'A';
char digit = '9';
char exclamation = '!';
```

You can use escape sequences to specify character constants, too:

```
char newline = '\n';
char tab = '\t';
char single_quote = '\'';
```

Of course, in every case the variable will be set to the code for the character between single quotes. The actual code value will depend on your computer environment, but by far the most common is American Standard Code for Information Interchange (ASCII). You can find the ASCII character set in Appendix B.

You can also initialize a variable of type char with an integer value, as long as the value fits into the range for type char with your compiler, as in this example:

```
char character = 74;    /* ASCII code for the letter J */
```

A variable of type char has a sort of dual personality: you can interpret it as a character or as an integer. Here's an example of an arithmetic operation with a value of type char:

```
char letter = 'C';      /* letter contains the decimal code value 67 */
letter = letter + 3;    /* letter now contains 70, which is 'F'      */
```

Thus, you can perform arithmetic on a value of type char and still treat it as a character.

Character Input and Character Output

You can read a single character from the keyboard and store it in a variable of type char using the scanf() function with the format specifier %c, for example

```
char ch = 0;
scanf("%c", &ch);   /* Read one character */
```

As you saw earlier, you must add an #include directive for the <stdio.h> header file to any source file in which you use the scanf() function:

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

To write a single character to the command line with the printf() function, you use the same format specifier, %c:

```
printf("The character is %c", ch);
```

Of course, you can output the numeric value of a character, too:

```
printf("The character is %c and the code value is %d", ch, ch);
```

This statement will output the value in ch as a character and as a numeric value.

TRY IT OUT: CHARACTER BUILDING

If you're completely new to programming, you may be wondering how on earth the computer knows whether it's dealing with a character or an integer. The reality is that it doesn't. It's a bit like when Alice encounters Humpty Dumpty who says "When I use a word, it means just what I choose it to mean—neither more nor less." An item of data in memory can mean whatever you choose it to mean. A byte containing the value 70 is a perfectly good integer. It's equally correct to regard it as the code for the letter J.

Let's look at an example that should make it clear. Here, you'll use the conversion specifier %c, which indicates that you want to output a value of type char as a character rather than an integer.

```
/* Program 2.15 Characters and numbers */
#include <stdio.h>

int main(void)
{
  char first = 'T';
  char second = 20;

  printf("\nThe first example as a letter looks like this - %c", first);
  printf("\nThe first example as a number looks like this - %d", first);
  printf("\nThe second example as a letter looks like this - %c", second);
  printf("\nThe second example as a number looks like this - %d\n", second);
  return 0;
}
```

The output from this program is the following:

```
The first example as a letter looks like this - T
The first example as a number looks like this - 84
The second example as a letter looks like this - ¶
The second example as a number looks like this - 20
```

How It Works

The program starts off by declaring two variables of type char:

```
char first = 'T';
char second = 20;
```

You initialize the first variable with a character constant and the second variable with an integer.

The next four statements output the value of each variable in two ways:

```
printf("\nThe first example as a letter looks like this - %c", first_example);
printf("\nThe first example as a number looks like this - %d", first_example);
printf("\nThe second example as a letter looks like this - %c", second_example);
printf("\nThe second example as a number looks like this - %d\n", second_example);
```

The %c conversion specifier interprets the contents of the variable as a single character, and the %d specifier interprets it as an integer. The numeric values that are output are the codes for the corresponding characters. These are ASCII codes in this instance, and will be in most instances, so that's what you'll assume throughout this book.

As I've noted, not all computers use the ASCII character set, so you may get different values than those shown previously. As long as you use the notation character for a character constant, you'll get the character that you want regardless of the character coding in effect.

You could also output the integer values of the variables of type `char` as hexadecimal values by using the format specifier `%x` instead of `%d`. You might like to try that.

TRY IT OUT: ARITHMETIC WITH VALUES THAT ARE CHARACTERS

Let's look at another example in which you apply arithmetic operations to values of type `char`:

```
/* Program 2.16 Using type char  */
#include <stdio.h>

int main(void)
{
  char first = 'A';
  char second = 'B';
  char last = 'Z';

  char number = 40;

  char ex1 = first + 2;             /* Add 2 to 'A'        */
  char ex2 = second - 1;            /* Subtract 1 from 'B' */
  char ex3 = last + 2;              /* Add 2 to 'Z'        */

  printf("Character values      %-5c%-5c%-5c", ex1, ex2, ex3);
  printf("\nNumerical equivalents %-5d%-5d%-5d", ex1, ex2, ex3);
  printf("\nThe number %d is the code for the character %c\n", number, number);
  return 0;
}
```

When you run the program you should get the following output:

```
Character values      C    A    \
Numerical equivalents 67   65   92
The number 40 is the code for the character (
```

How It Works

This program demonstrates how you can happily perform arithmetic with `char` variables that you've initialized with characters. The first three statements in the body of `main()` are as follows:

```
  char first = 'A';
  char second = 'B';
  char last = 'Z';
```

These initialize the variables `first`, `second`, and `last` to the character values you see. The numerical value of these variables will be the ASCII codes for the respective characters. Because you can treat them as numeric values as well as characters, you can perform arithmetic operations with them.

The next statement initializes a variable of type `char` with an integer value:

```
char number = 40;
```

The initializing value must be within the range of values that a 1-byte variable can store; so with my compiler, where `char` is a signed type, it must be between 128 and 127. Of course, you can interpret the contents of the variable as a character. In this case, it will be the character that has the ASCII code value 40, which happens to be a left parenthesis.

The next three statements declare three more variables of type `char`:

```
char ex1 = first + 2;              /* Add 2 to 'A'        */
char ex2 = second - 1;             /* Subtract 1 from 'B' */
char ex3 = last + 2;               /* Add 2 to 'Z'        */
```

These statements create new values and therefore new characters from the values stored in the variables `first`, `second`, and `last`; the results of these expressions are stored in the variables `ex1`, `ex2`, and `ex3`.

The next two statements output the three variables `ex1`, `ex2`, and `ex3` in two different ways:

```
printf("Character values      %-5c%-5c%-5c", ex1, ex2, ex3);
printf("\nNumerical equivalents %-5d%-5d%-5d", ex1, ex2, ex3);
```

The first statement interprets the values stored as characters by using the `%-5c` conversion specifier. This specifies that the value should be output as a character that is left-aligned in a field width of 5. The second statement outputs the same variables again, but this time interprets the values as integers by using the `%-5d` specifier. The alignment and the field width are the same but `d` specifies the output is an integer. You can see that the two lines of output show the three characters on the first line with their ASCII codes aligned on the line beneath.

The last line outputs the variable `number` as a character and as an integer:

```
printf("\nThe number %d is the code for the character %c\n", number, number);
```

To output the variable twice, you just write it twice—as the second and third arguments to the `printf()` function. It's output first as an integer value and then as a character.

This ability to perform arithmetic with characters can be very useful. For instance, to convert from uppercase to lowercase, you simply add the result of `‘a’-’A’` (which is 32 for ASCII) to the uppercase character. To achieve the reverse, just subtract `‘a’-’A’`. You can see how this works if you have a look at the decimal ASCII values for the alphabetic characters in Appendix B of this book. Of course, this operation depends on the character codes for `a` to `z` and `A` to `Z` being a contiguous sequence of integers. If this is not the case for the character coding used by your computer, this won't work. The EBCDIC code used on some IBM machines is an example of where you can't use this technique because there are discontinuities in the code values for letters.

The Wide Character Type

A variable of type `wchar_t` stores a multibyte character code and typically occupies 2 bytes. You would use type `wchar_t` when you are working with Unicode characters, for example. Type `wchar_t` is defined in the `<stddef.h>` standard header file, so you need to include this in source files that use this type. You can define a wide character constant by preceding what would otherwise be a character constant of type `char` with the modifier `L`. For example, here's how to declare a variable of type `wchar_t` and initialize it with the code for a capital `A`:

```
wchar_t w_ch = L'A';
```

Operations with type `wchar_t` work in much the same way as operations with type `char`. Since type `wchar_t` is an integer type, you can perform arithmetic operations with values of this type.

To read a character from the keyboard into a variable of type `wchar_t`, use the `%lc` format specification. Use the same format specifier to output a value of type `wchar_t`. Here's how you could read a character from the keyboard and then display it on the next line:

```
wchar_t wch = 0;
scanf("%lc", &wch);
printf("You entered %lc", wch);
```

Of course, you would need an `#include` directive for `<stdio.h>` for this fragment to compile correctly.

Enumerations

Situations arise quite frequently in programming when you want a variable that will store a value from a very limited set of possible values. One example is a variable that stores a value representing the current month in the year. You really would only want such a variable to be able to assume one of 12 possible values, corresponding to January through December. The enumeration in C is intended specifically for such purposes.

With an **enumeration** you can define a new integer type where variables of the type have a fixed range of possible values that you specify. Here's an example of a statement that defines a new type with the name `Weekday`:

```
enum Weekday {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
```

The name of the new type, `Weekday` in this instance, follows the `enum` keyword and this type name is referred to as the **tag** of the enumeration. Variables of type `Weekday` can have any of the values specified by the names that appear between the braces that follow the type name. These names are called **enumerators** or **enumeration constants** and there can be as many of these as you want. Each enumerator is identified by the unique name you assign, and the compiler will assign an integer value of type `int` to each name. An enumeration is an integer type because the enumerators that you specify will correspond to integer values that by default will start from zero with each successive enumerator having a value of one more than the previous enumerator. Thus, in this example, the values `Monday` through `Sunday` will map to values 0 through 6.

You could declare a new variable of type `Weekday` and initialize it like this:

```
enum Weekday today = Wednesday;
```

This declares a variable with the name `today` and it initializes it to the value `Wednesday`. Because the enumerators have default values, `Wednesday` will correspond to the value 2. The actual integer type that is used for a variable of an enumeration type is implementation-defined and the choice of type may depend on how many enumerators there are.

It is also possible to declare variables of the enumeration type when you define the type. Here's a statement that defines an enumeration type plus two variables:

```
enum Weekday {Monday, Tuesday, Wednesday, Thursday,
                             Friday, Saturday, Sunday} today, tomorrow;
```

This declares the enumeration type `Weekday` and two variables of that type, `today` and `tomorrow`.

Naturally you could also initialize the variable in the same statement so you could write this:

```
enum Weekday {Monday, Tuesday, Wednesday, Thursday,
                      Friday, Saturday, Sunday} today = Monday, tomorrow = Tuesday;
```

This initializes `today` and `tomorrow` to `Monday` and `Tuesday` respectively.

Because variables of an enumeration type are of an integer type, they can be used in arithmetic expressions. You could write the previous statement like this:

```
enum Weekday {Monday, Tuesday, Wednesday, Thursday,
              Friday, Saturday, Sunday} today = Monday, tomorrow = today + 1;
```

Now the initial value for `tomorrow` is one more than that of `today`. However, when you do this kind of thing, it is up to you to ensure that the value that results from the arithmetic is a valid enumerator value.

■ **Note** Although you specify a fixed set of possible values for an enumeration type, there is no checking mechanism to ensure that only these values are used in your program. It is up to you to make sure only valid enumeration values are used for a given enumeration type. One way to do this is to only assign values to variables of an enumeration type that are the enumeration constant names.

Choosing Enumerator Values

You can specify your own integer value for any or all of the enumerators explicitly. Although the names you use for enumerators must be unique, there is no requirement for the enumerator values themselves to be unique. Unless you have a specific reason for making some of the values the same, it is usually a good idea to ensure that they are unique. Here's how you could define the `Weekday` type so that the enumerator values start from 1:

```
enum Weekday {Monday=1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
```

Now the enumerators `Monday` through `Sunday` will correspond to values 1 through 7. The enumerators that follow an enumerator with an explicit value will be assigned successive integer values. This can cause enumerators to have duplicate values, as in the following example:

```
enum Weekday {Monday=5, Tuesday=4, Wednesday,
              Thursday=10, Friday =3, Saturday, Sunday};
```

`Monday`, `Tuesday`, `Thursday`, and `Friday` have explicit values specified. `Wednesday` will be set to `Tuesday+1` so it will be 5, the same as `Monday`. Similarly `Saturday` and `Sunday` will be set to 4 and 5 so they also have duplicate values. There's no reason why you can't do this, although unless you have a good reason for making some of the enumeration constants the same, it does tend to be confusing.

You can use an enumeration in any situation where you want a variable with a specific limited number of possible values. Here's another example of defining an enumeration:

```
enum Suit{clubs = 10, diamonds, hearts, spades);
enum Suit card_suit = diamonds;
```

The first statement defines the enumeration type `Suit`, so variables of this type can have one of the four values between the braces. The second statement defines a variable of type `Suit` and initializes it with the value `diamonds`, which will correspond to 11. You could also define an enumeration to identify card face values like this:

```
enum FaceValue { two=2, three, four, five, six, seven,
                 eight, nine, ten, jack, queen, king, ace};
```

In this enumeration the enumerators will have integer values that match the card value with `ace` as high.

When you output the value of a variable of an enumeration type, you'll just get the numeric value. If you want to output the enumerator name, you have to provide the program logic to do this. You'll be able to do this with what you learn in the next chapter.

Unnamed Enumeration Types

You can create variables of an enumeration type without specifying a tag, so there's no enumeration type name. For example

```
enum {red, orange, yellow, green, blue, indigo, violet} shirt_color;
```

There's no tag here so this statement defines an unnamed enumeration type with the possible enumerators from `red` to `violet`. The statement also declares one variable of the unnamed type with the name `shirt_color`.

You can assign a value to `shirt_color` in the normal way:

```
shirt_color = blue;
```

Obviously, the major limitation on unnamed enumeration types is that you must declare all the variables of the type in the statement that defines the type. Because you don't have a type name, there's no way to define additional variables of this type later in the code.

Variables to Store Boolean Values

The type `_Bool` stores Boolean values. A Boolean value typically arises from a comparison where the result may be true or false; you'll learn about comparisons and using the results to make decisions in your programs in Chapter 3. The value of a variable of type `_Bool` can be either 0 or 1, corresponding to the Boolean values `false` and `true` respectively, and because the values 0 and 1 are integers, type `_Bool` is regarded as an integer type. You declare a `_Bool` variable just like any other. For example

```
_Bool valid = 1;             /* Boolean variable initialized to true */
```

`_Bool` is not an ideal type name. The name `bool` would be less clumsy looking and more readable, but the Boolean type was introduced into the C language relatively recently so the type name was chosen to minimize the possibility of conflicts with existing code. If `bool` had been chosen as the type name, any program that used the name `bool` for some purpose most probably would not compile with a compiler that supported `bool` as a built-in type.

Having said that, you can use `bool` as the type name; you just need to add an `#include` directive for the standard header file `<stdbool.h>` to any source file that uses it. As well as defining `bool` to be the equivalent of `_Bool`, the `<stdbool.h>` header file also defines the symbols `true` and `false` to correspond to 1 and 0 respectively. Thus, if you include the header into your source file, you can rewrite the previous declaration as the following:

```
bool valid = true;           /* Boolean variable initialized to true */
```

This looks much clearer than the previous version so it's best to include the `<stdbool.h>` header unless you have a good reason not to.

You can cast between Boolean values and other numeric types. A nonzero numeric value will result in 1 (`true`) when cast to type `_Bool`, and 0 will cast to 0 (`false`). If you use a `_Bool` variable in an arithmetic expression, the compiler will insert an implicit conversion where necessary. Type `_Bool` has a rank lower than any of the other types, so in an operation involving type `_Bool` and a value of another type it is the `_Bool` value that will be converted to the other type.

I won't elaborate further on working with Boolean variables at this point. You'll learn more about using them in the next chapter.

The Complex Number Types

This section assumes you have learned about complex numbers at some point. If you have never heard of complex numbers, you can safely skip this section. In case you are a little rusty on complex numbers, I'll remind you of their basic characteristics.

A complex number is a number of the form `a + bi` (or `a + bj` if you are an electrical engineer) where `i` is the square root of minus one, and `a` and `b` are real numbers. `a` is the real part, and `bi` is the imaginary part of the complex number. A complex number can also be regarded as an ordered pair of real numbers `(a, b)`.

Complex numbers can be represented in the complex plane, as illustrated in Figure 2-2.

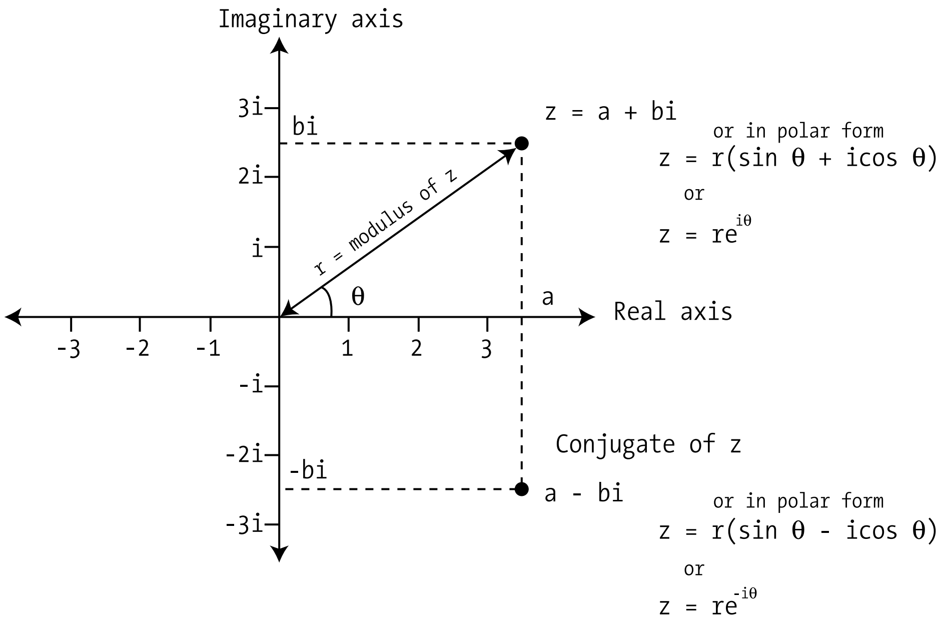

Figure 2-2. *Representing a complex number in the complex plane*

You can apply the following operations to complex numbers:

- *Modulus*: The modulus of a complex number `a + bi` is √ `(a2 + b2)`.
- *Equality*: The complex numbers `a + bi` and `c + di` are equal if `a` equals `c` and `b` equals `d`.
- *Addition*: The sum of the complex numbers `a + bi` and `c + di` is `(a + c) + (b + d)i`.
- *Multiplication*: The product of the complex numbers `a + bi` and `c + di` is `(ac - bd) + (ad + bc)i`.
- *Division*: The result of dividing the complex number `a + bi` by `c + di` is `(ac - bd) / (c2 + d2) + ((bc - ad)(c2 + d2))i`.
- *Conjugate*: The conjugate of a complex number `a + bi` is `a - bi`. Note that the product of a complex number `a + bi` and its conjugate is `a2 + b2`.

Complex numbers also have a **polar representation**: a complex number can be written in polar form as `r(sin θ+ icos θ)` or as the ordered pair of real numbers `(r,θ)` where `r` and `θ` are as shown in Figure 2-2. From Euler's formula a complex number can also be represented as `reiθ`.

I'll just briefly introduce the idea of the types in the C language that store complex numbers because the applications for these are very specialized. You have three types that store complex numbers:

- `float _Complex` with real and imaginary parts of type `float`
- `double _Complex` with real and imaginary parts of type `double`
- `long double _Complex` with real and imaginary parts of type `long double`

You could declare a variable to store complex numbers like this:

```
double _Complex z1;             /* Real and imaginary parts are type double */
```

The somewhat cumbersome `_Complex` keyword was chosen for the complex number types for the same reasons as type `_Bool`: to avoid breaking existing code. But the `<complex.h>` header defines `complex` as being equivalent to `_Complex`, as well as many other functions and macros for working with complex numbers. With the `<complex.h>` header included into the source file, you can use `complex` instead of `_Complex`, so you could declare the variable `z1` like this:

```
double complex z1;              /* Real and imaginary parts are type double */
```

The imaginary unit, which is the square root of 1, is represented by the keyword `_Complex_I`, notionally as a value of type `float`. Thus you can write a complex number with the real part as 2.0 and the imaginary part as 3.0 as `2.0 + 3.0 * _Complex_I`. The `<complex.h>` header defines `I` to be the equivalent of `_Complex_I`, so you can use this much simpler representation as long as you have included the header in your source file. Thus you can write the previous example of a complex number as `2.0 + 3.0 * I`. You could therefore declare and initialize the variable `z1` with this statement:

```
double complex z1 = 2.0 + 3.0*I;  /* Real and imaginary parts are type double */
```

The `creal()` function returns the real part of a value of type `double complex` that is passed as the argument, and `cimag()` returns the imaginary part. For example

```
double real_part = creal(z1);     /* Get the real part of z1 */
double imag_part = cimag(z1);     /* Get the imaginary part of z1 */
```

You append an `f` to these function names when you are working with `float complex` values (`crealf()` and `cimagf()`) and a lowercase `L` when you are working with `long double complex` values (`creall()` and `cimagl()`). The `conj()` function returns the complex conjugate of its `double complex` argument, and you have the `conjf()` and `conjl()` functions for the other two complex types.

You use the `_Imaginary` keyword to define variables that store purely imaginary numbers; in other words there is no real component. There are three types for imaginary numbers, using the keywords `float`, `double`, and `long double`, analogous to the three complex types. The `<complex.h>` header defines `imaginary` as a more readable equivalent of `_Imaginary`, so you could declare a variable that stores imaginary numbers like this:

```
double imaginary ix = 2.4*I;
```

Casting an imaginary value to a complex type produces a complex number with a zero real part and a complex part the same as the imaginary number. Casting a value of an imaginary type to a real type other than `_Bool` results in 0. Casting a value of an imaginary type to type `_Bool` results in 0 for a zero imaginary value, and 1 otherwise.

You can write arithmetic expressions involving complex and imaginary values using the arithmetic operators `+`, , `*`, and `/`. Let's see them at work.

TRY IT OUT: WORKING WITH COMPLEX NUMBERS

Here’s a simple example that creates a couple of complex variables and performs some simple arithmetic operations:

```
/* Program 2.17 Working with complex numbers
#include <complex.h>
#include <stdio.h>

int main(void)
{
  double complex cx = 1.0 + 3.0*I;
  double complex  cy = 1.0 - 4.0*I;
  printf("Working with complex numbers:");
  printf("\nStarting values: cx = %.2f%+.2fi  cy = %.2f%+.2fi",
                                 creal(cx), cimag(cx), creal(cy), cimag(cy));

  double complex  sum = cx+cy;
  printf("\n\nThe sum cx + cy = %.2f%+.2fi",
                                         creal(sum),cimag(sum));

  double complex  difference = cx-cy;
  printf("\n\nThe difference cx - cy = %.2f%+.2fi",
                                        creal(difference),cimag(difference));

  double complex product = cx*cy;
  printf("\n\nThe product cx * cy = %.2f%+.2fi",
                                              creal(product),cimag(product));

  double complex quotient = cx/cy;
  printf("\n\nThe quotient cx / cy = %.2f%+.2fi",
                                            creal(quotient),cimag(quotient));

  double complex conjugate = conj(cx);
  printf("\n\nThe conjugate of cx =  %.2f%+.2fi",
                                         creal(conjugate) ,cimag(conjugate));
    return 0;
}
```

You should get the following output from this example:

```
Working with complex numbers:
Starting values: cx = 1.00+3.00i  cy = 1.00-4.00i

The sum cx + cy = 2.00-1.00i

The difference cx - cy = 0.00+7.00i

The product cx * cy = 13.00-1.00i

The quotient cx / cy = -0.65+0.41i

The conjugate of cx =  1.00-3.00i
```

How It Works

The code is fairly self-explanatory. After defining and initializing the variables `cx` and `cy`, you use the four arithmetic operators with these, and output the result in each case. You could equally well use the keyword `_Complex` instead of `complex` if you wish.

The output specification used for the imaginary part of each complex value is `%+.2f`. The `+` following the `%` specifies that the sign should always be output. If the `+` was omitted you would only get the sign in the output when the value is negative. The `2` following the decimal point specifies that two places after the decimal point are to be output.

If you explore the contents of the `<complex.h>` header that is supplied with your compiler you'll find it provides a wide range of other functions that operate on complex values.

The op= Form of Assignment

C is fundamentally a very concise language, so it provides you with abbreviated shortcuts for some operations. Consider the following line of code:

```
number = number + 10;
```

This sort of assignment, in which you're incrementing or decrementing a variable by some amount occurs very often so there's a shorthand version:

```
number += 10;
```

The `+=` operator after the variable name is one example of a family of `op=` operators. This statement has exactly the same effect as the previous one and it saves a bit of typing. The `op` in `op=` can be any of the arithmetic operators:

```
+  -  *  /  %
```

If you suppose `number` has the value 10, you can write the following statements:

```
number *= 3;           /* number will be set to number*3 which is 30 */
number /= 3;           /* number will be set to number/3 which is 3  */
number %= 3;           /* number will be set to number%3 which is 1  */
```

The `op` in `op=` can also be a few other operators that you haven't encountered yet:

```
 <<  >>  &  ^  |
```

I'll defer discussion of these to Chapter 3, however.

The `op=` set of operators always works in the same way. If you have a statement of the form

```
lhs op= rhs;
```

where `rhs` represents any expression on the right-hand side of the `op=` operator, then the effect is the same as a statement of the form

```
lhs = lhs op (rhs);
```

Note the parentheses around the `rhs` expression. This means that `op` applies to the value that results from evaluating the entire `rhs` expression, whatever it is. So just to reinforce your understanding of this, let's look at few more examples. The statement

```
variable *= 12;
```

is the same as

```
variable = variable * 12;
```

You now have two different ways of incrementing an integer variable by one. Both of the following statements increment `count` by 1:

```
count = count +1;
count += 1;
```

You'll learn about yet another way of doing this in the next chapter. This amazing level of choice tends to make it virtually impossible for indecisive individuals to write programs in C.

Because the `op` in `op=` applies to the result of evaluating the `rhs` expression, the statement

```
a /= b+1;
```

is the same as

```
a = a/(b+1);
```

Your computational facilities have been somewhat constrained so far. You've been able to use only a very basic set of arithmetic operators. You can get more power to your calculating elbow using standard library facilities, so before you come to the final example in this chapter, you'll take a look at some of the mathematical functions that the standard library offers.

Mathematical Functions

The `math.h` header file includes declarations for a wide range of mathematical functions. To give you a feel for what's available, you'll take a look at those that are used most frequently. All the functions return a value of type `double`.

You have the set of functions shown in Table 2-9 available for numerical calculations of various kinds. These all require arguments to be of type `double`.

Table 2-9. *Functions for Numerical Calculations*

Function	Operation
`floor(x)`	Returns the largest integer that isn't greater than `x` as type `double`
`ceil(x)`	Returns the smallest integer that isn't less than `x` as type `double`
`fabs(x)`	Returns the absolute value of `x`
`log(x)`	Returns the natural logarithm (base e) of `x`
`log10(x)`	Returns the logarithm to base 10 of `x`
`exp(x)`	Returns the value of e^x
`sqrt(x)`	Returns the square root of `x`
`pow(x)`	Returns the value x^y

Here are some examples of using these functions:

```
double x = 2.25;
double less = 0.0;
double more = 0.0;
double root = 0.0;
less = floor(x);   /* Result is 2.0 */
more = ceil(x);    /* Result is 3.0 */
root = sqrt(x);    /* Result is 1.5 */
```

You also have a range of trigonometric functions available, as shown in Table 2-10. Arguments and values returned are again of type `double` and angles are expressed in radians.

Table 2-10. *Functions for Trigonometry*

Function	Operation
`sin(x)`	Sine of `x` expressed in radians
`cos(x)`	Cosine of `x`
`tan(x)`	Tangent of `x`

If you're into trigonometry, the use of these functions will be fairly self-evident. Here are some examples:

```
double angle = 45.0;           /* Angle in degrees */
double pi = 3.14159265;
double sine = 0.0;
double cosine = 0.0;
sine = sin(pi*angle/180.0);    /*Angle converted to radians */
cosine = sin(pi*angle/180.0);  /*Angle converted to radians */
```

Because 180 degrees is the same angle as radians, dividing an angle measured in degrees by 180 and multiplying by the value of will produce the angle in radians, as required by these functions.

You also have the inverse trigonometric functions available: `asin()`, `acos()`, and `atan()`, as well as the hyperbolic functions `sinh()`, `cosh()`, and `tanh()`. Don't forget, you must include `math.h` into your program if you wish to use any of these functions. If this stuff is not your bag, you can safely ignore this section.

Designing a Program

Now it's time for the end-of-chapter real-life example. It would be a great idea to try out some of the numeric types in a new program. I'll take you through the basic elements of the process of writing a program from scratch. This involves receiving an initial specification of the problem, analyzing the problem, preparing a solution, writing the program, and, of course, running the program and testing it to make sure it works. Each step in the process can introduce problems, beyond just the theory.

The Problem

The height of a tree is of great interest to many people. For one thing, if a tree is being cut down, knowing its height tells you how far away *safe* is. This is very important to those with a nervous disposition. Your problem is to find out the height of a tree without using a very long ladder, which itself would introduce risk to life and limb. To find the height of a tree, you're allowed the help of a friend—preferably a short friend. You should assume that the tree you're measuring is taller than both you and your friend. Trees that are shorter than you present little risk, unless they're of the spiky kind.

The Analysis

Real-world problems are rarely expressed in terms that are directly suitable for programming. Before you consider writing a line of code, you need to be sure that you have a complete understanding of the problem and how it's going to be solved. Only then can you estimate how much time and effort will be involved in creating the solution.

The analysis phase involves gaining a full understanding of the problem and determining the logical process for solving it. Typically this requires a significant amount of work. It involves teasing out any detail in the specification of the problem that is vague or missing. Only when you fully understand the problem can you begin to express the solution in a form that's suitable for programming.

You're going to determine the height of a tree using some simple geometry and the heights of two people: you and one other. Let's start by naming the tall person (you) Lofty and the shorter person (your friend) Shorty. If you're vertically challenged, the roles can be reversed. For more accurate results, the tall person should be significantly taller than the short person. Otherwise the tall person could consider standing on a box. The diagram in Figure 2-3 will give you an idea of what you're trying to do in this program.

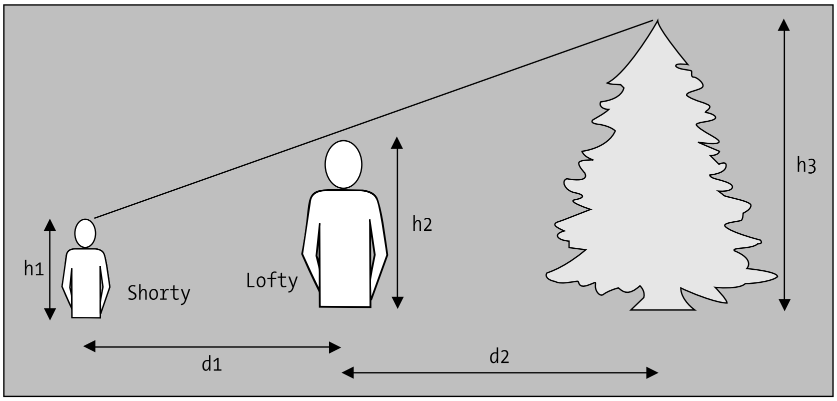

Figure 2-3. *The height of a tree*

Finding the height of the tree is actually quite simple. You can get the height of the tree, h_3, if you know the other dimensions shown in the illustration: h_1 and h_2, which are the heights of Shorty and Lofty, and d_1 and d_2, which are the distances between Shorty and Lofty and Lofty and the tree, respectively. You can use the technique of similar triangles to work out the height of the tree. You can see this in the simplified diagram in Figure 2-4.

Here, because the triangles are similar, height1 divided by distance1 is equal to height2 divided by distance2. Using this relationship, you can get the height of the tree from the height of Shorty and Lofty and the distances to the tree, as shown in Figure 2-5.

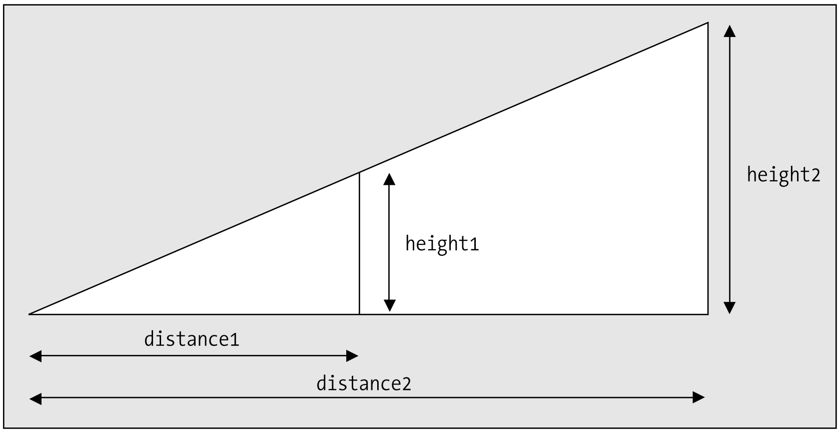

Figure 2-4. *Similar triangles*

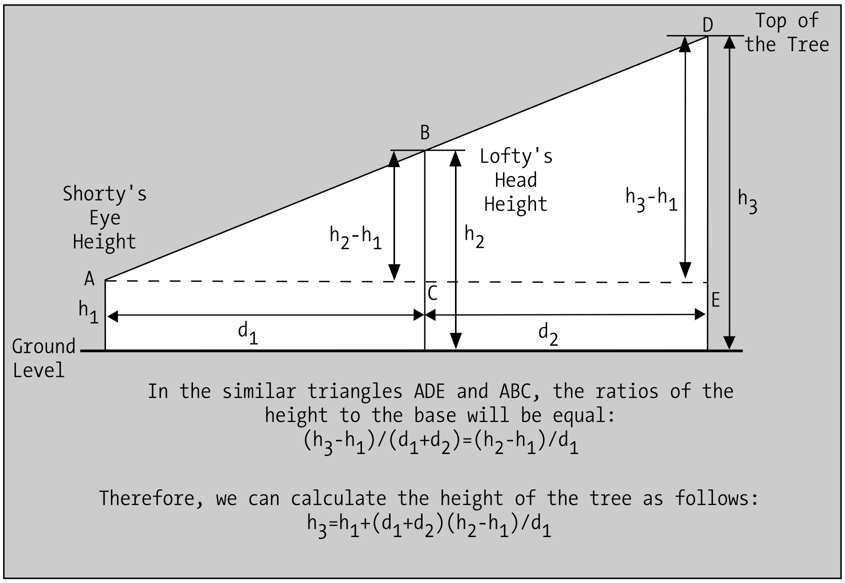
D
Top of the Tree
B
Lofty's Head Height
Shorty's Eye Height
h_3-h_1
h_3
h_2-h_1
h_2
A
C
E
h_1
d_1
d_2
Ground Level
In the similar triangles ADE and ABC, the ratios of the height to the base will be equal:
$(h_3-h_1)/(d_1+d_2)=(h_2-h_1)/d_1$
Therefore, we can calculate the height of the tree as follows:
$h_3=h_1+(d_1+d_2)(h_2-h_1)/d_1$

Figure 2-5. *Calculating the tree height*

The triangles ADE and ABC are the same as those shown in Figure 2-4. Using the fact that the triangles are similar, you can calculate the height of the tree as shown in the equation at the bottom of Figure 2-5.

This means that you can calculate the height of the tree in your program from four values:

- The distance between Shorty and Lofty, d_1 in the diagram. You'll use the variable `shorty_to_lofty` to store this value.
- The distance between Lofty and the tree, d_2 in the diagram. You'll use the variable `lofty_to_tree` to store this value.

- The height of Lofty to the top of his head, h_2 in the diagram. You'll use the variable `lofty` to store this value.
- The height of Shorty, but only up to the eyes, h_1 in the diagram. You'll use the variable `shorty` to store this value.

You can then plug these values into the equation for the height of the tree.

Your first task is to get these four values into the computer. You can then use your ratios to find out the height of the tree and finally output the answer. The steps are as follows:

1. Input the values you need.
2. Calculate the height of the tree using the equation in the diagram.
3. Display the answer.

The Solution

This section outlines the steps you'll take to solve the problem.

Step 1

Your first step is to get the values that you need to work out the height of the tree. This means that you have to include the `stdio.h` header file, because you need to use both `printf()` and `scanf()`. You then have to decide what variables you need to store these values in. After that, you can use `printf()` to prompt for the input and `scanf()` to read the values from the keyboard.

You'll provide for the heights of the participants to be entered in feet and inches for the convenience of the user. Inside the program, though, it will be easier to work with all heights and distances in the same units, so you'll convert all measurements to inches. You'll need two variables to store the heights of Shorty and Lofty in inches. You'll also need a variable to store the distance between Lofty and Shorty, and another to store the distance from Lofty to the tree—both distances in inches, of course.

In the input process, you'll first get Lofty's height as a number of whole feet and then as a number of inches, prompting for each value as you go along. You can use two more variables for this: one to store the feet value and the other to store the inches value. You'll then convert these into just inches and store the result in the variable you've reserved for Lofty's height. You'll do the same thing for Shorty's height (but only up to the height of his or her eyes) and finally the same for the distance between them. For the distance to the tree, you'll use only whole feet, because this will be accurate enough—and again you'll convert the distance to inches. You can reuse the same variables for each measurement in feet and inches that is entered. So here goes with the first part of the program:

```
/* Program 2.18  Calculating the height of a tree */
#include <stdio.h>

int main(void)
{
  long shorty = 0L;          /* Shorty's height in inches                */
  long lofty = 0L;           /* Lofty's height in inches                 */
  long feet = 0L;
  long inches = 0L;
  long shorty_to_lofty = 0L; /* Distance from Shorty to Lofty in inches   */
  long lofty_to_tree = 0L;   /* Distance from Lofty to the tree in inches */
  const long inches_per_foot = 12L;
  /* Get Lofty's height */
  printf("Enter Lofty's height to the top of his/her head, in whole feet: ");
  scanf("%ld", &feet);
  printf("              ...and then inches: ");
  scanf("%ld", &inches);
  lofty = feet*inches_per_foot + inches;

  /* Get Shorty's height up to his/her eyes */
  printf("Enter Shorty's height up to his/her eyes, in whole feet: ");
  scanf("%ld", &feet);
  printf("                               ... and then inches: ");
  scanf("%ld", &inches);
  shorty = feet*inches_per_foot + inches;

  /* Get the distance from Shorty to Lofty */
  printf("Enter the distance between Shorty and Lofty, in whole feet: ");
  scanf("%ld", &feet);
  printf("                                        ... and then inches: ");
  scanf("%ld", &inches);
  shorty_to_lofty = feet*inches_per_foot + inches;

  /* Get the distance from Lofty to the tree */
  printf("Finally enter the distance to the tree to the nearest foot: ");
   scanf("%ld", &feet);
  lofty_to_tree = feet*inches_per_foot;

  /* The code to calculate the height of the tree will go here */

  /* The code to display the result will go here               */
  return 0;
}
```

Notice how the program code is spaced out to make it easier to read. You don't have to do it this way, but if you decide to change the program next year, it will make it much easier to see how the program works if it's well laid out. You should always add comments to your programs to help with this. It's particularly important to at least make clear what the variables are used for and to document the basic logic of the program.

You use a variable that you've declared as `const` to convert from feet to inches. The variable name, `inches_per_foot`, makes it reasonably obvious what's happening when it's used in the code. This is much better than using the "magic number" 12 explicitly. Here you're dealing with feet and inches, and most people will be aware that there are 12 inches in a foot. In other circumstances the significance of numeric constants may not be so obvious, though. If you're using the value 0.22 in a program calculating salaries, it's much less apparent what this might be; therefore, the calculation may seem rather obscure. If you create a `const` variable `tax_rate` that you've initialized to 0.22 and use that instead, then the mist clears.

Step 2

Now that you have all the data you need, you can calculate the height of the tree. All you need to do is implement the equation for the tree height in terms of your variables. You'll need to declare another variable to store the height of the tree.

You can now add the code that's shown here in bold type to do this:

```
/* Program 2.18  Calculating the height of a tree */
#include <stdio.h>

int main(void)
{
  long shorty = 0L;          /* Shorty's height in inches                 */
  long lofty = 0L;           /* Lofty's height in inches                  */
  long feet = 0L;            /* A whole number of feet                    */
  long inches = 0L;
  long shorty_to_lofty = 0;  /* Distance from Shorty to Lofty in inches   */
  long lofty_to_tree = 0;    /* Distance from Lofty to the tree in inches */
  long tree_height = 0;     /* Height of the tree in inches            */
  const long inches_per_foot = 12L;

  /* Get Lofty's height */
  printf("Enter Lofty's height to the top of his/her head, in whole feet: ");
  scanf("%ld", &feet);
  printf("              ...and then inches: ");
  scanf("%ld", &inches);
  lofty = feet*inches_per_foot + inches;

  /* Get Shorty's height up to his/her eyes */
  printf("Enter Shorty's height up to his/her eyes, in whole feet: ");
  scanf("%ld", &feet);
   printf("                               ... and then inches: ");
  scanf("%ld", &inches);
  shorty = feet*inches_per_foot + inches;

  /* Get the distance from Shorty to Lofty */
  printf("Enter the distance between Shorty and Lofty, in whole feet: ");
  scanf("%ld", &feet);
  printf("                                        ... and then inches: ");
  scanf("%ld", &inches);
  shorty_to_lofty = feet*inches_per_foot + inches;

  /* Get the distance from Lofty to the tree */
  printf("Finally enter the distance to the tree to the nearest foot: ");
  scanf("%ld", &feet);
  lofty_to_tree = feet*inches_per_foot;

  /* Calculate the height of the tree in inches */
  tree_height = shorty + (shorty_to_lofty + lofty_to_tree)*(lofty-shorty)/
                                                                shorty_to_lofty;

  /* The code to display the result will go here              */
  return 0;
}
```

The statement to calculate the height is essentially the same as the equation in the diagram. It's a bit messy, but it translates directly to the statement in the program to calculate the height.

Step 3

Finally, you need to output the answer. To present the result in the most easily understandable form, you'll convert the result that you've stored in `tree_height`—which is in inches—back into feet and inches:

```
/* Program 2.18  Calculating the height of a tree */
#include <stdio.h>

int main(void)
{
  long shorty = 0L;          /* Shorty's height in inches                 */
  long lofty = 0L;           /* Lofty's height in inches                  */
  long feet = 0L;
  long inches = 0L;
  long shorty_to_lofty = 0;  /* Distance from Shorty to Lofty in inches   */
  long lofty_to_tree = 0;    /* Distance from Lofty to the tree in inches */
  long tree_height = 0;     /* Height of the tree in inches            */
  const long inches_per_foot = 12L;

  /* Get Lofty's height */
  printf("Enter Lofty's height to the top of his/her head, in whole feet: ");
  scanf("%ld", &feet);
  printf("                                         ... and then inches: ");
  scanf("%ld", &inches);
  lofty = feet*inches_per_foot + inches;

  /* Get Shorty's height up to his/her eyes */
  printf("Enter Shorty's height up to his/her eyes, in whole feet: ");
  scanf("%ld", &feet);
  printf("                                  ... and then inches: ");
  scanf("%ld", &inches);
  shorty = feet*inches_per_foot + inches;

  /* Get the distance from Shorty to Lofty */
  printf("Enter the distance between Shorty and Lofty, in whole feet: ");
  scanf("%ld", &feet);
  printf("                                      ... and then inches: ");
  scanf("%ld", &inches);
  shorty_to_lofty = feet*inches_per_foot + inches;

  /* Get the distance from Lofty to the tree */
  printf("Finally enter the distance to the tree to the nearest foot: ");
  scanf("%ld", &feet);
  lofty_to_tree = feet*inches_per_foot;

  /* Calculate the height of the tree in inches */
  tree_height = shorty + (shorty_to_lofty + lofty_to_tree)*(lofty-shorty)/
                                                              shorty_to_lofty;

  /* Display the result in feet and inches              */
  printf("The height of the tree is %ld feet and %ld inches.\n",
                    tree_height/inches_per_foot, tree_height% inches_per_foot);
return 0;
}
```

And there you have it. The output from the program looks something like this:

```
Enter Lofty's height to the top of his/her head, in whole feet first: 6
                                                 ... and then inches: 2
Enter Shorty's height up to his/her eyes, in whole feet: 4
                                    ... and then inches: 6
Enter the distance between Shorty and Lofty, in whole feet : 5
                                        ... and then inches: 0
Finally enter the distance to the tree to the nearest foot: 20
The height of the tree is 12 feet and 10 inches.
```

Summary

This chapter covered quite a lot of ground. By now, you know how a C program is structured, and you should be fairly comfortable with any kind of arithmetic calculation. You should also be able to choose variable types to suit the job at hand. Aside from arithmetic, you've added quite a bit of input and output capability to your knowledge. You should now feel at ease with inputting values into variables via `scanf()`. You can output text and the values of character and numeric variables to the screen. You won't remember it all the first time around, but you can always look back over this chapter if you need to. Not bad for the first two chapters, is it?

In the next chapter, you'll start looking at how you can control the program by making decisions depending on the values you enter. As you can probably imagine, this is key to creating interesting and professional programs.

Table 2-11 summarizes the real variable types you've used so far. You can look back at these when you need a reminder as you continue through the book.

Table 2-11. *Variable Types and Value Ranges*

Type	Number of Bytes	Range of Values
`char`	1	128 to +127 or 0 to +255
`unsigned char`	1	0 to +255
`short`	2	32,768 to +32,767
`unsigned short`	2	0 to +65,535
`int`	4	32,768 to +32,767 or 2,147,438,648 to +2,147,438,647
`unsigned int`	4	0 to +65,535 or 0 to +4,294,967,295
`long`	4	2,147,438,648 to +2,147,438,647
`unsigned long`	4	0 to +4,294,967,295
`long long`	8	9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
`unsigned long long`	8	0 to +18,446,744,073,709,551,615
`float`	4	±3.4E38 (6 digits)
`double`	8	±1.7E308 (15 digits)
`long double`	12	±1.2E4932 (19 digits)

The types that store complex data are shown in Table 2-12.

Table 2-12. *Complex Types*

Type	Description
`float _Complex`	Stores a complex number with real and imaginary parts as type `float`
`double _Complex`	Stores a complex number with real and imaginary parts as type `double`
`long double _Complex`	Stores a complex number with real and imaginary parts as type `long double`
`float _Imaginary`	Stores an imaginary number as type `float`
`double _Imaginary`	Stores an imaginary number as type `double`
`long double _Imaginary`	Stores an imaginary number as type `long double`

The `<complex.h>` header file defines `complex` and `imaginary` as alternatives to the keywords `_Complex` and `_Imaginary` and it defines `I` to represent, `i`, the square root of 1.

You have seen and used some of the data output format specifications with the `printf()` function in this chapter and you'll find the complete set described in Appendix D. Appendix D also describes the input format specifiers that you use to control how data is interpreted when it's read from the keyboard by the `scanf()` function. Whenever you are unsure about how you deal with a particular kind of data for input or output, just look in Appendix D.

Exercises

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Source Code/Download section of the Apress web site (`http://www.apress.com`), but that really should be a last resort.

Exercise 2-1. Write a program that prompts the user to enter a distance in inches and then outputs that distance in yards, feet, and inches.

Exercise 2-2. Write a program that prompts for input of the length and width of a room in feet and inches, and then calculates and outputs the floor area in square yards with two decimal places after the decimal point.

Exercise 2-3. You're selling a product that's available in two versions: type 1 is a standard version priced at $3.50, and type 2 is a deluxe version priced at $5.50. Write a program using only what you've learned up to now that prompts for the user to enter the product type and a quantity, and then calculates and outputs the price for the quantity entered.

Exercise 2-4. Write a program that prompts for the user's weekly pay in dollars and the hours worked to be entered through the keyboard as floating-point values. The program should then calculate and output the average pay per hour in the following form:

```
Your average hourly pay rate is 7 dollars and 54 cents.
```