2

Introducing C

You will learn about the following in this chapter:

- Operator:

  `=`
- Functions:

  `main(), printf()`
- Putting together a simple C program
- Creating integer-valued variables, assigning them values, and displaying those values onscreen
- The newline character
- How to include comments in your programs, create programs containing more than one function, and find program errors
- What keywords are

What does a C program look like? If you skim through this book, you'll see many examples. Quite likely, you'll find that C looks a little peculiar, sprinkled with symbols such as `{, cp->tort`, and `*ptr++`. As you read through this book, however, you will find that the appearance of these and other characteristic C symbols grows less strange, more familiar, and perhaps even welcome! Or, if you already are familiar with one of C's many descendants, you might feel as if you are coming home to the source. In this chapter, we begin by presenting a simple sample program and explaining what it does. At the same time, we highlight some of C's basic features.

A Simple Example of C

Let's take a look at a simple C program. This program, shown in Listing 2.1, serves to point out some of the basic features of programming in C. Before you read the upcoming line-by-line explanation of the program, read through Listing 2.1 to see whether you can figure out for yourself what it will do.

Listing 2.1 **The `first.c` Program**

```
#include <stdio.h>
int main(void)                /* a simple program             */
{
    int num;                  /* define a variable called num */
    num = 1;                  /* assign a value to num        */

    printf("I am a simple "); /* use the printf() function    */
    printf("computer.\n");
    printf("My favorite number is %d because it is first.\n",num);

    return 0;
}
```

If you think this program will print something on your screen, you're right! Exactly what will be printed might not be apparent, so run the program and see the results. First, use your favorite editor (or your compiler's favorite editor) to create a file containing the text from Listing 2.1. Give the file a name that ends in `.c` and that satisfies your local system's name requirements. You can use `first.c`, for example. Now compile and run the program. (Check Chapter 1, "Getting Ready," for some general guidelines to this process.) If all went well, the output should look like the following:

```
I am a simple computer.
My favorite number is 1 because it is first.
```

All in all, this result is not too surprising, but what happened to the `\n`s and the `%d` in the program? And some of the lines in the program do look strange. It's time for an explanation.

> **Program Adjustments**
>
> Did the output for this program briefly flash onscreen and then disappear? Some windowing environments run the program in a separate window and then automatically close the window when the program finishes. In this case, you can supply extra code to make the window stay open until you strike a key. One way is to add the following line before the return statement:
>
> ```
> getchar();
> ```
>
> This code causes the program to wait for a keystroke, so the window remains open until you press a key. You'll learn more about `getchar()` in Chapter 8, "Character Input/Output and Input Validation."

The Example Explained

We'll take two passes through the program's source code. The first pass ("Pass 1: Quick Synopsis") highlights the meaning of each line to help you get a general feel for what's going

on. The second pass ("Pass 2: Program Details") explores specific implications and details to help you gain a deeper understanding.

Figure 2.1 summarizes the parts of a C program; it includes more elements than our first example uses.

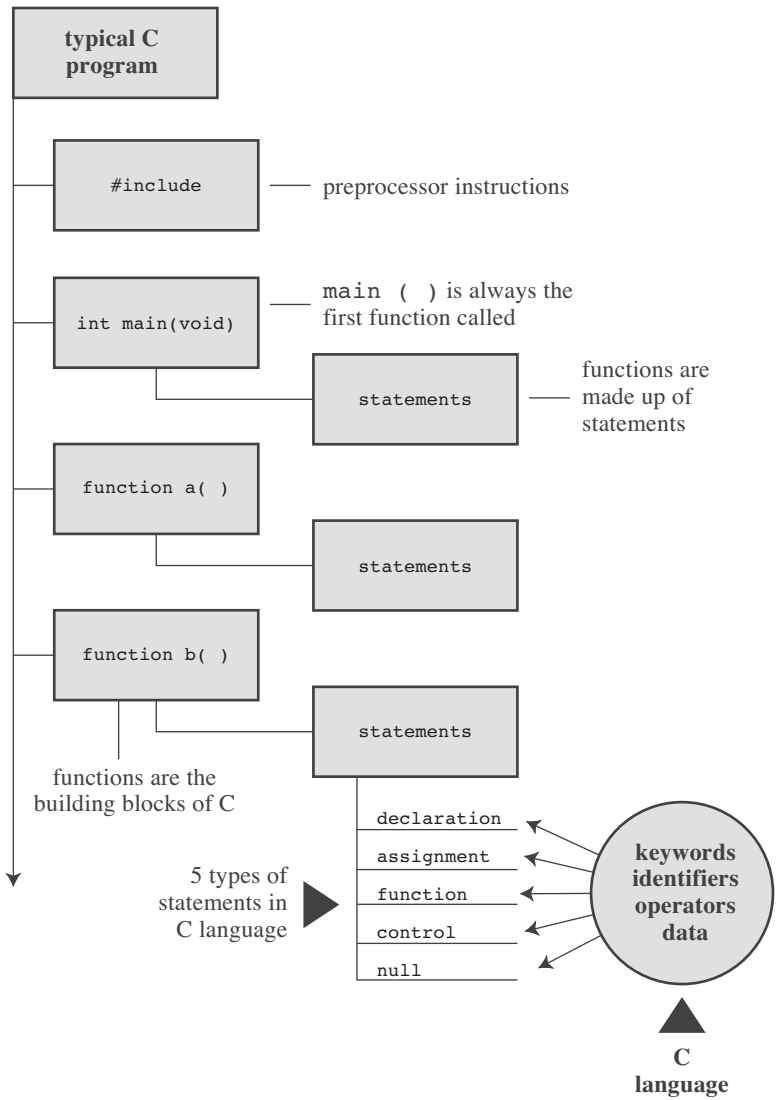

Figure 2.1 Anatomy of a C program.

Pass 1: Quick Synopsis

This section presents each line from the program followed by a short description; the next section (Pass 2) explores the topics raised here more fully.

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

This line tells the compiler to include the information found in the file `stdio.h`, which is a standard part of all C compiler packages; this file provides support for keyboard input and for displaying output.

```
int main(void)        ←a function name
```

C programs consist of one or more *function*s, the basic modules of a C program. This program consists of one function called `main`. The parentheses identify `main()` as a function name. The `int` indicates that the `main()` function returns an integer, and the `void` indicates that `main()` doesn't take any arguments. These are matters we'll go into later. Right now, just accept both `int` and `void` as part of the standard ANSI C way for defining `main()`. (If you have a pre-ANSI C compiler, omit `void`; you may want to get something more recent to avoid incompatibilities.)

```
/* a simple program */    ←a comment
```

The symbols `/*` and `*/` enclose comments—remarks that help clarify a program. They are intended for the reader only and are ignored by the compiler.

```
{        ←beginning of the body of the function
```

This opening brace marks the start of the statements that make up the function. A closing brace (`}`) marks the end of the function definition.

```
int num;        ←a declaration statement
```

This statement announces that you are using a variable called `num` and that `num` will be an `int` (integer) type.

```
num = 1;    ←an assignment statement
```

The statement `num = 1;` assigns the value `1` to the variable called `num`.

```
printf("I am a simple ");    ←a function call statement
```

The first statement using `printf()` displays the phrase `I am a simple` on your screen, leaving the cursor on the same line. Here `printf()` is part of the standard C library. It's termed a *function*, and using a function in the program is termed *calling a function*.

```
printf("computer.\n");    ←another function call statement
```

The next call to the `printf()` function tacks on `computer` to the end of the last phrase printed. The `\n` is code telling the computer to start a new line—that is, to move the cursor to the beginning of the next line.

```
printf("My favorite number is %d because it is first.\n", num);
```

The last use of `printf()` prints the value of `num` (which is `1`) embedded in the phrase in quotes. The `%d` instructs the computer where and in what form to print the value of `num`.

```
return 0;    ←a return statement
```

A C function can furnish, or *return*, a number to the agency that used it. For the present, just regard this line as the appropriate closing for a `main()` function.

```
}    ←the end
```

As promised, the program ends with a closing brace.

Pass 2: Program Details

Now that you have an overview of Listing 2.1, we'll take a closer look. Once again, we'll examine the individual lines from the program, this time using each line of code as a starting point for going deeper into the details behind the code and as a basis for developing a more general perspective of C programming features.

`#include` Directives and Header Files

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

This is the line that begins the program. The effect of `#include <stdio.h>` is the same as if you had typed the entire contents of the `stdio.h` file into your file at the point where the `#include` line appears. In effect, it's a cut-and-paste operation. `include` files provide a convenient way to share information that is common to many programs.

The `#include` statement is an example of a C *preprocessor directive*. In general, C compilers perform some preparatory work on source code before compiling; this is termed *preprocessing*.

The `stdio.h` file is supplied as part of all C compiler packages. It contains information about input and output functions, such as `printf()`, for the compiler to use. The name stands for *standard input/output header*. C people call a collection of information that goes at the top of a file a *header*, and C implementations typically come with several header files.

For the most part, header files contain information used by the compiler to build the final executable program. For example, they may define constants or indicate the names of functions and how they should be used. But the actual code for a function is in a library file of precompiled code, not in a header file. The linker component of the compiler takes care of finding the library code you need. In short, header files help guide the compiler in putting your program together correctly.

ANSI/ISO C has standardized which header files a C compiler must make available. Some programs need to include `stdio.h`, and some don't. The documentation for a particular C implementation should include a description of the functions in the C library. These function descriptions identify which header files are needed. For example, the description for `printf()` says to use `stdio.h`. Omitting the proper header file might not affect a particular program, but

it is best not to rely on that. Each time this book uses library functions, it will use the `include` files specified by the ANSI/ISO standard for those functions.

> **Note Why Input and Output Are Not Built In**
>
> Perhaps you are wondering why facilities as basic as input and output aren't included automatically. One answer is that not all programs use this I/O (input/output) package, and part of the C philosophy is to avoid carrying unnecessary weight. This principle of economic use of resources makes C popular for embedded programming—for example, writing code for a chip that controls an automotive fuel system or a Blu-ray player. Incidentally, the `#include` line is not even a C language statement! The # symbol in column 1 identifies the line as one to be handled by the C preprocessor before the compiler takes over. You will encounter more examples of preprocessor instructions later, and Chapter 16, "The C Preprocessor and the C Library," discusses this topic more fully.

The `main()` Function

```
int main(void)
```

This next line from the program proclaims a function by the name of `main`. True, `main` is a rather plain name, but it is the only choice available. A C program (with some exceptions we won't worry about) always begins execution with the function called `main()`. You are free to choose names for other functions you use, but `main()` must be there to start things. What about the parentheses? They identify `main()` as a function. You will learn more about functions soon. For now, just remember that functions are the basic modules of a C program.

The `int` is the `main()` function's return type. That means that the kind of value `main()` can return is an integer. Return where? To the operating system—we'll come back to this question in Chapter 6, "C Control Statements: Looping."

The parentheses following a function name generally enclose information being passed along to the function. For this simple example, nothing is being passed along, so the parentheses contain the word `void`. (Chapter 11, "Character Strings and String Functions," introduces a second format that allows information to be passed to `main()` from the operating system.)

If you browse through ancient C code, you'll often see programs starting off with the following format:

```
main()
```

The C90 standard grudgingly tolerated this form, but the C99 and C11 standards don't. So even if your current compiler lets you do this, don't.

The following is another form you may see:

```
void main()
```

Some compilers allow this, but none of the standards have ever listed it as a recognized option. Therefore, compilers don't have to accept this form, and several don't. Again, stick to the

standard form, and you won't run into problems if you move a program from one compiler to another.

Comments

```
/* a simple program */
```

The parts of the program enclosed in the `/* */` symbols are comments. Using comments makes it easier for someone (including yourself) to understand your program. One nice feature of C comments is that they can be placed anywhere, even on the same line as the material they explain. A longer comment can be placed on its own line or even spread over more than one line. Everything between the opening `/*` and the closing `*/` is ignored by the compiler. The following are some valid and invalid comment forms:

```
/* This is a C comment. */
/* This comment, being somewhat wordy, is spread over
   two lines. */
/*
  You can do this, too.
*/
/* But this is invalid because there is no end marker.
```

C99 added a second style of comments, one popularized by C++ and Java. The new style uses the symbols `//` to create comments that are confined to a single line:

```
// Here is a comment confined to one line.
int rigue;      // Such comments can go here, too.
```

Because the end of the line marks the end of the comment, this style needs comment markers just at the beginning of the comment.

The newer form is a response to a potential problem with the old form. Suppose you have the following code:

```
/*
I hope this works.
*/
x = 100;
y = 200;
/* Now for something else. */
```

Next, suppose you decide to remove the fourth line and accidentally delete the third line (the `*/`), too. The code then becomes

```
/*
I hope this works.
y = 200;
/* Now for something else. */
```

Now the compiler pairs the `/*` in the first line with the `*/` in the fourth line, making all four lines into one comment, including the line that was supposed to be part of the code. Because the `//` form doesn't extend over more than one line, it can't lead to this "disappearing code" problem.

Some compilers may not support this feature; others may require changing a compiler setting to enable C99 or C11 features.

This book, operating on the theory that needless consistency can be boring, uses both kinds of comments.

Braces, Bodies, and Blocks

```
{
...
}
```

In Listing 2.1, braces delimited the `main()` function. In general, all C functions use braces to mark the beginning as well as the end of the body of a function. Their presence is mandatory, so don't leave them out. Only braces (`{ }`) work for this purpose, not parentheses (`( )`) and not brackets (`[ ]`).

Braces can also be used to gather statements within a function into a unit or block. If you are familiar with Pascal, ADA, Modula-2, or Algol, you will recognize the braces as being similar to `begin` and `end` in those languages.

Declarations

```
int num;
```

This line from the program is termed a *declaration statement*. The declaration statement is one of C's most important features. This particular example declares two things. First, somewhere in the function, you have a *variable* called `num`. Second, the `int` proclaims `num` as an integer—that is, a number without a decimal point or fractional part. (`int` is an example of a *data type*.) The compiler uses this information to arrange for suitable storage space in memory for the `num` variable. The semicolon at the end of the line identifies the line as a C *statement* or instruction. The semicolon is part of the statement, not just a separator between statements as it is in Pascal.

The word `int` is a C *keyword* identifying one of the basic C data types. Keywords are the words used to express a language, and you can't use them for other purposes. For instance, you can't use `int` as the name of a function or a variable. These keyword restrictions don't apply outside the language, however, so it is okay to name a cat or favorite child `int`. (Local custom or law may void this option in some locales.)

The word `num` in this example is an *identifier*—that is, a name you select for a variable, a function, or some other entity. So the declaration connects a particular identifier with a particular location in computer memory, and it also establishes the type of information, or data type, to be stored at that location.

In C, *all* variables must be declared *before* they are used. This means that you have to provide lists of all the variables you use in a program and that you have to show which data type each variable is. Declaring variables is considered a good programming technique, and, in C, it is mandatory.

Traditionally, C has required that variables be declared at the beginning of a block with no other kind of statement allowed to come before any of the declarations. That is, the body of `main()` might look like the following:

```
int main()    // traditional rules
{
    int doors;
    int dogs;
    doors = 5;
    dogs = 3;
    // other statements
}
```

C99 and C11, following the practice of C++, let you place declarations about anywhere in a block. However, you still must declare a variable before its first use. So if your compiler supports this feature, your code can look like the following:

```
int main()          // current C rules
{
// some statements
    int doors;
    doors = 5;      // first use of doors
// more statements
    int dogs;
    dogs = 3;       // first use of dogs
    // other statements
}
```

For greater compatibility with older systems, this book will stick to the original convention.

At this point, you probably have three questions. First, what are data types? Second, what choices do you have in selecting a name? Third, why do you have to declare variables at all? Let's look at some answers.

Data Types

C deals with several kinds (or types) of data: integers, characters, and floating point, for example. Declaring a variable to be an integer or a character type makes it possible for the computer to store, fetch, and interpret the data properly. You'll investigate the variety of available types in the next chapter.

Name Choice

You should use meaningful names (or identifiers) for variables (such as `sheep_count` instead of `x3` if your program counts sheep). If the name doesn't suffice, use comments to explain what the variables represent. Documenting a program in this manner is one of the basic techniques of good programming.

With C99 and C11 you can make the name of an identifier as long as you want, but the compiler need only consider the first 63 characters as significant. For external identifiers (see Chapter 12, "Storage Classes, Linkage, and Memory Management") only 31 characters need to be recognized. This is a substantial increase from the C90 requirement of 31 characters and six characters, respectively, and older C compilers often stopped at eight characters max. Actually, you can use more than the maximum number of characters, but the compiler isn't required to pay attention to the extra characters. What does this mean? If you have two identifiers each 63 characters long and identical except for one character, the compiler is required to recognize them as distinct from one another. If you have two identifiers 64 characters long and identical except for the final character, the compiler might recognize them as distinct, or it might not; the standard doesn't define what should happen in that case.

The characters at your disposal are lowercase letters, uppercase letters, digits, and the underscore (_). The first character must be a letter or an underscore. The following are some examples:

Valid Names	Invalid Names
`wiggles`	`$Z]**`
`cat2`	`2cat`
`Hot_Tub`	`Hot-Tub`
`taxRate`	`tax rate`
`_kcab`	`don't`

Operating systems and the C library often use identifiers with one or two initial underscore characters, such as in `_kcab`, so it is better to avoid that usage yourself. The standard labels beginning with one or two underscore characters, such as library identifiers, are *reserved*. This means that although it is not a syntax error to use them, it could lead to name conflicts.

C names are *case sensitive*, meaning an uppercase letter is considered distinct from the corresponding lowercase letter. Therefore, `stars` is different from `Stars` and `STARS`.

To make C more international, C99 and C11 make an extensive set of characters available for use by the Universal Character Names (or *UMC*) mechanism. Reference Section VII, "Expanded Character Support," in Appendix B discusses this addition. This makes available characters that are not part of the English alphabet.

Four Good Reasons to Declare Variables

Some older languages, such as the original forms of FORTRAN and BASIC, allow you to use variables without declaring them. So why can't you take this easy-going approach in C? Here are some reasons:

- Putting all the variables in one place makes it easier for a reader to grasp what the program is about. This is particularly true if you give your variables meaningful names (such as `taxrate` instead of `r`). If the name doesn't suffice, use comments to explain what the variables represent. Documenting a program in this manner is one of the basic techniques of good programming.
- Thinking about which variables to declare encourages you to do some planning before plunging into writing a program. What information does the program need to get started? What exactly do I want the program to produce as output? What is the best way to represent the data?
- Declaring variables helps prevent one of programming's more subtle and hard-to-find bugs—that of the misspelled variable name. For example, suppose that in some language that lacks declarations, you made the statement

  ```
  RADIUS1 = 20.4;
  ```

  and that elsewhere in the program you mistyped

  ```
  CIRCUM = 6.28 * RADIUSl;
  ```

  You unwittingly replaced the numeral 1 with the letter *l* (lowercase el). That other language would create a new variable called `RADIUSl` and use whatever value it had (perhaps zero, perhaps garbage). `CIRCUM` would be given the wrong value, and you might have a heck of a time trying to find out why. This can't happen in C (unless you were silly enough to declare two such similar variable names) because the compiler will complain when the undeclared `RADIUSl` shows up.
- Your C program will not compile if you don't declare your variables. If the preceding reasons fail to move you, you should give this one serious thought.

Given that you need to declare your variables, where do they go? As mentioned before, C prior to C99 required that the declarations go at the beginning of a block. A good reason for following this practice is that grouping the declarations together makes it easier to see what the program is doing. Of course, there's also a good reason to spread your declarations around, as C99 now allows. The idea is to declare variables just before you're ready to give them a value. That makes it harder to forget to give them a value. As a practical matter, many compilers don't yet support the C99 rule.

Assignment

```
num = 1;
```

The next program line is an *assignment statement*, one of the basic operations in C. This particular example means "assign the value `1` to the variable `num`." The earlier `int num;` line set aside

space in computer memory for the variable `num`, and the assignment line stores a value in that location. You can assign `num` a different value later, if you want; that is why `num` is termed a *variable*. Note that the assignment statement assigns a value from the right side to the left side. Also, the statement is completed with a semicolon, as shown in Figure 2.2.

Figure 2.2 The assignment statement is one of the basic C operations.

The `printf()` Function

```
printf("I am a simple ");
printf("computer.\n");
printf("My favorite number is %d because it is first.\n", num);
```

These lines all use a standard C function called `printf()`. The parentheses signify that `printf` is a function name. The material enclosed in the parentheses is information passed from the `main()` function to the `printf()` function. For example, the first line passes the phrase `I am a simple` to the `printf()` function. Such information is called the *argument* or, more fully, the *actual argument* of a function (see Figure 2.3). (C uses the terms *actual argument* and *formal argument* to distinguish between a specific value sent to a function and a variable in the function used to hold the value; Chapter 5 "Operators, Expressions, and Statements," goes into this matter in more detail.) What does the function `printf()` do with this argument? It looks at whatever lies between the double quotation marks and prints that text onscreen.

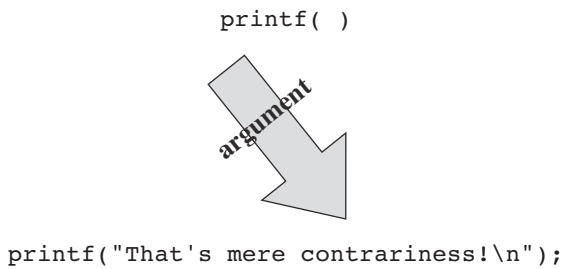

Figure 2.3 The `printf()` function with an argument.

This first `printf()` line is an example of how you *call* or *invoke* a function in C. You need type only the name of the function, placing the desired argument(s) within the parentheses. When the program reaches this line, control is turned over to the named function (`printf()` in this case). When the function is finished with whatever it does, control is returned to the original (the *calling*) function—`main()`, in this example.

What about this next `printf()` line? It has the characters `\n` included in the quotes, and they didn't get printed! What's going on? The `\n` symbol means to start a new line. The `\n` combination (typed as two characters) represents a single character called the *newline character*. To `printf()`, it means "start a new line at the far-left margin." In other words, printing the newline character performs the same function as pressing the Enter key of a typical keyboard. Why not just use the Enter key when typing the `printf()` argument? That would be interpreted as an immediate command to your editor, not as an instruction to be stored in your source code. In other words, when you press the Enter key, the editor quits the current line on which you are working and starts a new one. The newline character, however, affects how the output of the program is displayed.

The newline character is an example of an *escape sequence*. An escape sequence is used to represent difficult- or impossible-to-type characters. Other examples are `\t` for Tab and `\b` for Backspace. In each case, the escape sequence begins with the backslash character, `\`. We'll return to this subject in Chapter 3, "Data and C."

Well, that explains why the three `printf()` statements produced only two lines: The first print instruction didn't have a newline character in it, but the second and third did.

The final `printf()` line brings up another oddity: What happened to the `%d` when the line was printed? As you will recall, the output for this line was

```
My favorite number is 1 because it is first.
```

Aha! The digit `1` was substituted for the symbol group `%d` when the line was printed, and `1` was the value of the variable `num`. The `%d` is a placeholder to show where the value of `num` is to be printed. This line is similar to the following BASIC statement:

```
PRINT "My favorite number is "; num; " because it is first."
```

The C version does a little more than this, actually. The `%` alerts the program that a variable is to be printed at that location, and the `d` tells it to print the variable as a decimal (base 10) integer. The `printf()` function allows several choices for the format of printed variables, including hexadecimal (base 16) integers and numbers with decimal points. Indeed, the `f` in `printf()` is a reminder that this is a *formatting* print function. Each type of data has its own specifier—as the book introduces new types, it will also introduce the appropriate specifiers.

Return Statement

```
return 0;
```

This return statement is the final statement of the program. The `int` in `int main(void)` means that the `main()` function is supposed to return an integer. The C standard requires that `main()` behave that way. C functions that return values do so with a return statement, which consists of the keyword `return`, followed by the returned value, followed by a semicolon. If you leave out the return statement for `main()`, the program will return 0 when it reaches the closing `}`. So you can omit the return statement at the end of `main()`. However, you can't omit it from other functions, so it's more consistent to use it in `main()`, too. At this point, you can regard the return statement in `main()` as something required for logical consistency, but it has a practical use with some operating systems, including Linux and Unix. Chapter 11 will deal further with this topic.

The Structure of a Simple Program

Now that you've seen a specific example, you are ready for a few general rules about C programs. A *program* consists of a collection of one or more functions, one of which must be called `main()`. The description of a *function* consists of a header and a body. The *function header* contains the function name along with information about the type of information passed to the function and returned by the function. You can recognize a function name by the parentheses, which may be empty. The *body* is enclosed by braces (`{}`) and consists of a series of statements, each terminated by a semicolon (see Figure 2.4). The example in this chapter had a *declaration statement*, announcing the name and type of variable being used. Then it had an *assignment statement* giving the variable a value. Next, there were three *print statements*, each calling the `printf()` function. The print statements are examples of *function call statements*. Finally, `main()` ends with a *return statement*.

In short, a simple standard C program should use the following format:

```
#include <stdio.h>
int main(void)
{
   statements
   return 0;
}
```

(Recall that each statement includes a terminating semicolon.)

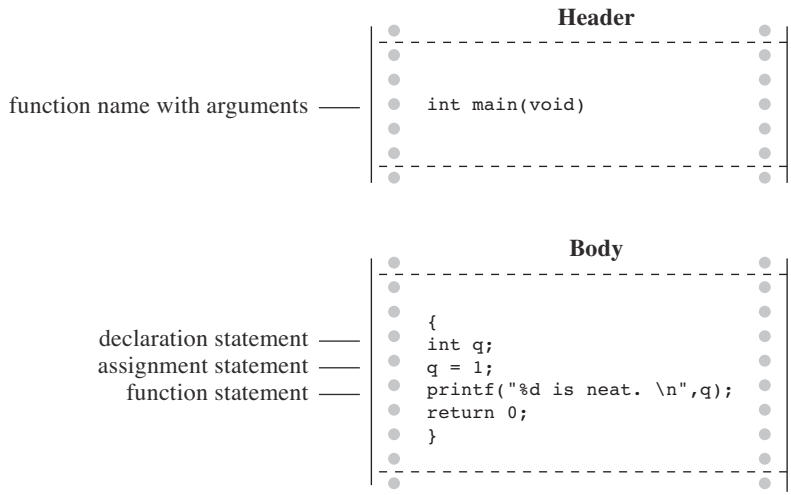

Figure 2.4 A function has a header and a body.

Tips on Making Your Programs Readable

Making your programs readable is good programming practice. A readable program is much easier to understand, and that makes it easier to correct or modify. The act of making a program readable also helps clarify your own concept of what the program does.

You've already seen two techniques for improving readability: Choose meaningful variable names and use comments. Note that these two techniques complement each other. If you give a variable the name `width`, you don't need a comment saying that this variable represents a width, but a variable called `video_routine_4` begs for an explanation of what video routine 4 does.

Another technique involves using blank lines to separate one conceptual section of a function from another. For example, the simple sample program has a blank line separating the declaration section from the action section. C doesn't require the blank line, but it enhances readability.

A fourth technique is to use one line per statement. Again, this is a readability convention, not a C requirement. C has a *free-form* format. You can place several statements on one line or spread one statement over several. The following is legitimate, but ugly, code:

```
int main(  void  ) { int four; four
=
4
;
printf(
      "%d\n",
four); return 0;}
```

The semicolons tell the compiler where one statement ends and the next begins, but the program logic is much clearer if you follow the conventions used in this chapter's example (see Figure 2.5).

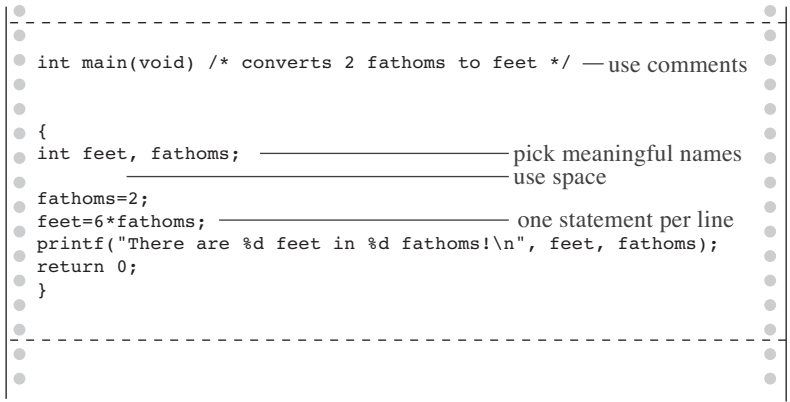

Figure 2.5 Making your program readable.

Taking Another Step in Using C

The first sample program was pretty easy, and the next example, shown in Listing 2.2, isn't much harder.

Listing 2.2 **The `fathm_ft.c` Program**

```
// fathm_ft.c -- converts 2 fathoms to feet

#include <stdio.h>
int main(void)
{
    int feet, fathoms;

    fathoms = 2;
    feet = 6 * fathoms;
    printf("There are %d feet in %d fathoms!\n", feet, fathoms);
    printf("Yes, I said %d feet!\n", 6 * fathoms);

    return 0;
}
```

What's new? The code provides a program description, declares multiple variables, does some multiplication, and prints the values of two variables. Let's examine these points in more detail.

Documentation

First, the program begins with a comment (using the new comment style) identifying the filename and the purpose of the program. This kind of program documentation takes but a moment to do and is helpful later when you browse through several files or print them.

Multiple Declarations

Next, the program declares two variables instead of just one in a single declaration statement. To do this, separate the two variables (`feet` and `fathoms`) by a comma in the declaration statement. That is,

```
int feet, fathoms;
```

and

```
int feet;
int fathoms;
```

are equivalent.

Multiplication

Third, the program makes a calculation. It harnesses the tremendous computational power of a computer system to multiply 2 by 6. In C, as in many languages, `*` is the symbol for multiplication. Therefore, the statement

```
feet = 6 * fathoms;
```

means "look up the value of the variable `fathoms`, multiply it by 6, and assign the result of this calculation to the variable `feet`."

Printing Multiple Values

Finally, the program makes fancier use of `printf()`. If you compile and run the example, the output should look like this:

```
There are 12 feet in 2 fathoms!
Yes, I said 12 feet!
```

This time, the code made *two* substitutions in the first use of `printf()`. The first `%d` in the quotes was replaced by the value of the first variable (`feet`) in the list following the quoted segment, and the second `%d` was replaced by the value of the second variable (`fathoms`) in the

list. Note that the list of variables to be printed comes at the tail end of the statement after the quoted part. Also note that each item is separated from the others by a comma.

The second use of `printf()` illustrates that the value printed doesn't have to be a variable; it just has to be something, such as `6 * fathoms`, that reduces to a value of the right type.

This program is limited in scope, but it could form the nucleus of a program for converting fathoms to feet. All that is needed is a way to assign additional values to `feet` interactively; we will explain how to do that in later chapters.

While You're at It—Multiple Functions

So far, these programs have used the standard `printf()` function. Listing 2.3 shows you how to incorporate a function of your own—besides `main()`—into a program.

Listing 2.3 **The `two_func.c` Program**

```
//* two_func.c -- a program using two functions in one file */
#include <stdio.h>
void butler(void);      /* ANSI/ISO C function prototyping */
int main(void)
{
    printf("I will summon the butler function.\n");
    butler();
    printf("Yes. Bring me some tea and writeable DVDs.\n");

    return 0;
}

void butler(void)       /* start of function definition */
{
    printf("You rang, sir?\n");
}
```

The output looks like the following:

```
I will summon the butler function.
You rang, sir?
Yes. Bring me some tea and writeable DVDs.
```

The `butler()` function appears three times in this program. The first appearance is in the *prototype*, which informs the compiler about the functions to be used. The second appearance is in `main()` in the form of a *function call*. Finally, the program presents the *function definition*, which is the source code for the function itself. Let's look at each of these three appearances in turn.

The C90 standard added prototypes, and older compilers might not recognize them. (We'll tell you what to do when using such compilers in a moment.) A prototype declares to the compiler that you are using a particular function, so it's called a *function declaration*. It also specifies properties of the function. For example, the first `void` in the prototype for the `butler()` function indicates that `butler()` does not have a return value. (In general, a function can return a value to the calling function for its use, but `butler()` doesn't.) The second `void`—the one in `butler(void)`—means that the `butler()` function has no arguments. Therefore, when the compiler reaches the point in `main()` where `butler()` is used, it can check to see whether `butler()` is used correctly. Note that `void` is used to mean "empty," not "invalid."

Older C supported a more limited form of function declaration in which you just specified the return type but omitted describing the arguments:

```
void butler();
```

Older C code uses function declarations like the preceding one instead of function prototypes. The C90, C99, and C11 standards recognize this older form but indicate it will be phased out in time, so don't use it. If you inherit some legacy C code, you may want to convert the old-style declarations to prototypes. Later chapters in this book return to prototyping, function declarations, and return values.

Next, you invoke `butler()` in `main()` simply by giving its name, including parentheses. When `butler()` finishes its work, the program moves to the next statement in `main()`.

Finally, the function `butler()` is defined in the same manner as `main()`, with a function header and the body enclosed in braces. The header repeats the information given in the prototype: `butler()` takes no arguments and has no return value. For older compilers, omit the second `void`.

One point to note is that it is the location of the `butler()` call in `main()`—not the location of the `butler()` definition in the file—that determines when the `butler()` function is executed. You could, for example, put the `butler()` definition above the `main()` definition in this program, and the program would still run the same, with the `butler()` function executed between the two calls to `printf()` in `main()`. Remember, all C programs begin execution with `main()`, no matter where `main()` is located in the program files. However, the usual C practice is to list `main()` first because it normally provides the basic framework for a program.

The C standard recommends that you provide function prototypes for all functions you use. The standard `include` files take care of this task for the standard library functions. For example, under standard C, the `stdio.h` file has a function prototype for `printf()`. The final example in Chapter 6 will show you how to extend prototyping to non-`void` functions, and Chapter 9 covers functions fully.

Introducing Debugging

Now that you can write a simple C program, you are in a position to make simple errors. Program errors often are called *bug*s, and finding and fixing the errors is called *debugging*. Listing 2.4 presents a program with some bugs. See how many you can spot.

Listing 2.4 **The `nogood.c` Program**

```
/*  nogood.c -- a program with errors */
#include <stdio.h>
int main(void)
(
    int n, int n2, int n3;

/* this program has several errors
    n = 5;
    n2 = n * n;
    n3 = n2 * n2;
    printf("n = %d, n squared = %d, n cubed = %d\n", n, n2, n3)

    return 0;
)
```

Syntax Errors

Listing 2.4 contains several syntax errors. You commit a *syntax error* when you don't follow C's rules. It's analogous to a grammatical error in English. For instance, consider the following sentence: *Bugs frustrate be can*. This sentence uses valid English words but doesn't follow the rules for word order, and it doesn't have quite the right words, anyway. C syntax errors use valid C symbols in the wrong places.

So what syntax errors did `nogood.c` make? First, it uses parentheses instead of braces to mark the body of the function—it uses a valid C symbol in the wrong place. Second, the declaration should have been

```
int n, n2, n3;
```

or perhaps

```
int n;
int n2;
int n3;
```

Next, the example omits the `*/` symbol pair necessary to complete a comment. (Alternatively, you could replace `/*` with the new `//` form.) Finally, it omits the mandatory semicolon that should terminate the `printf()` statement.

How do you detect syntax errors? First, before compiling, you can look through the source code and see whether you spot anything obvious. Second, you can examine errors found by the compiler because part of its job is to detect syntax errors. When you attempt to compile this program, the compiler reports back any errors it finds, identifying the nature and location of each error.

However, the compiler can get confused. A true syntax error in one location might cause the compiler to mistakenly think it has found other errors. For instance, because the example does not declare `n2` and `n3` correctly, the compiler might think it has found further errors whenever those variables are used. In fact, if you can't make sense of all the reported errors, rather than trying to correct all the reported errors at once, you should correct just the first one or two and then recompile; some of the other errors may go away. Continue in this way until the program works. Another common compiler trick is reporting the error a line late. For instance, the compiler may not deduce that a semicolon is missing until it tries to compile the next line. So if the compiler complains of a missing semicolon on a line that has one, check the line before.

Semantic Errors

Semantic errors are errors in meaning. For example, consider the following sentence: *Scornful derivatives sing greenly*. The syntax is fine because adjectives, nouns, verbs, and adverbs are in the right places, but the sentence doesn't mean anything. In C, you commit a semantic error when you follow the rules of C correctly but to an incorrect end. The example has one such error:

```
n3 = n2 * n2;
```

Here, `n3` is supposed to represent the cube of `n`, but the code sets it up to be the fourth power of `n`.

The compiler does not detect semantic errors, because they don't violate C rules. The compiler has no way of divining your true intentions. That leaves it to you to find these kinds of errors. One way is to compare what a program does to what you expected it to do. For instance, suppose you fix the syntax errors in the example so that it now reads as shown in Listing 2.5.

Listing 2.5 **The `stillbad.c` Program**

```
/* stillbad.c -- a program with its syntax errors fixed */
#include <stdio.h>
int main(void)
{
    int n, n2, n3;

/* this program has a semantic error */
    n = 5;
    n2 = n * n;
    n3 = n2 * n2;
    printf("n = %d, n squared = %d, n cubed = %d\n", n, n2, n3);
```

```
    return 0;
}
```

Its output is this:

```
n = 5, n squared = 25, n cubed = 625
```

If you are cube-wise, you'll notice that 625 is the wrong value. The next stage is to track down how you wound up with this answer. For this example, you probably can spot the error by inspection. In general, however, you need to take a more systematic approach. One method is to pretend you are the computer and to follow the program steps one by one. Let's try that method now.

The body of the program starts by declaring three variables: `n`, `n2`, and `n3`. You can simulate this situation by drawing three boxes and labeling them with the variable names (see Figure 2.6). Next, the program assigns `5` to `n`. Simulate that by writing `5` into the `n` box. Next, the program multiplies `n` by `n` and assigns the result to `n2`, so look in the `n` box, see that the value is `5`, multiply `5` by `5` to get `25`, and place `25` in box `n2`. To duplicate the next C statement (`n3 = n2 * n2;`), look in `n2` and find `25`. You multiply `25` by `25`, get `625`, and place it in `n3`. Aha! You are squaring `n2` instead of multiplying it by `n`.

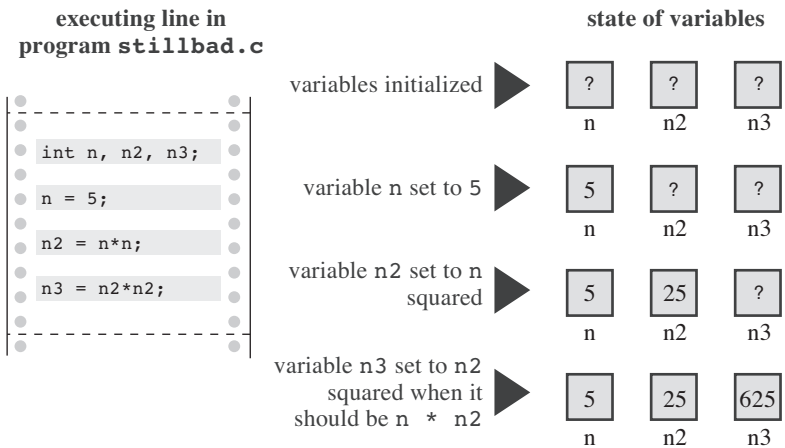

Figure 2.6 Tracing a program.

Well, perhaps this procedure is overkill for this example, but going through a program step-by-step in this fashion is often the best way to see what's happening.

Program State

By tracing the program step-by-step manually, keeping track of each variable, you monitor the program state. The *program state* is simply the set of values of all the variables at a given point in program execution. It is a snapshot of the current state of computation.

We just discussed one method of tracing the state: executing the program step-by-step yourself. In a program that makes, say, 10,000 iterations, you might not feel up to that task. Still, you can go through a few iterations to see whether your program does what you intend. However, there is always the possibility that you will execute the steps as you intended them to be executed instead of as you actually wrote them, so try to be faithful to the actual code.

Another approach to locating semantic problems is to sprinkle extra `printf()` statements throughout to monitor the values of selected variables at key points in the program. Seeing how the values change can illuminate what's happening. After you have the program working to your satisfaction, you can remove the extra statements and recompile.

A third method for examining the program states is to use a debugger. A *debugger* is a program that enables you to run another program step-by-step and examine the value of that program's variables. Debuggers come in various levels of ease of use and sophistication. The more advanced debuggers show which line of source code is being executed. This is particularly handy for programs with alternative paths of execution because it is easy to see which particular paths are being followed. If your compiler comes with a debugger, take time now to learn how to use it. Try it with Listing 2.4, for example.

Keywords and Reserved Identifiers

Keywords are the vocabulary of C. Because they are special to C, you can't use them as identifiers, for example, or as variable names. Many of these keywords specify various types, such as `int`. Others, such as `if`, are used to control the order in which program statements are executed. In the following list of C keywords, boldface indicates keywords added by the C90 standard, italics indicates new keywords added by the C99 standard, and boldface italics indicates keywords added by the C11 standard.

ISO C Keywords

`auto`	`extern`	`short`	`while`
`break`	`float`	**`signed`**	***`_Alignas`***
`case`	`for`	`sizeof`	***`_Alignof`***
`char`	`goto`	`static`	***`_Bool`***
`const`	`if`	`struct`	***`_Complex`***
`continue`	*`inline`*	`switch`	***`_Generic`***
`default`	`int`	`typedef`	***`_Imaginary`***

ISO C Keywords

do	long	union	*_Noreturn*
double	register	unsigned	*_Static_assert*
else	restrict	void	*#_Thread_local*
enum	return	**volatile**	

If you try to use a keyword, for, say, the name of a variable, the compiler catches that as a syntax error. There are other identifiers, called *reserved identifiers*, that you shouldn't use. They don't cause syntax errors because they are valid names. However, the language already uses them or reserves the right to use them, so it could cause problems if you use these identifiers to mean something else. Reserved identifiers include those beginning with an underscore character and the names of the standard library functions, such as `printf()`.

Key Concepts

Computer programming is a challenging activity. It demands abstract, conceptual thinking combined with careful attention to detail. You'll find that compilers enforce the attention to detail. When you talk to a friend, you might use a few words incorrectly, make a grammatical error or two, perhaps leave some sentences unfinished, yet your friend will still understand what you are trying to say. But a compiler doesn't make such allowances; to it, almost right is still wrong.

The compiler won't help you with conceptual matters, such as these, so this book will try to fill that gap by outlining the key concepts in each chapter.

For this chapter, your goal should be to understand what a C program is. You can think of a program as a description you prepare of how you want the computer to behave. The compiler handles the really detailed job of converting your description to the underlying machine language. (As a measure of how much work a compiler does, it can create an executable file of 60KB from your source code file of 1KB; a lot of machine language goes into representing even a simple C program.) Because the compiler has no real intelligence, you have to express your description in the compiler's terms, and these terms are the formal rules set up by the C language standard. (Although restrictive, this still is far better than having to express your description directly in machine language!)

The compiler expects to receive its instructions in a specific format, which we described in detail in this chapter. Your job as a programmer is to express your ideas about how a program should behave within the framework that the compiler—guided by the C standard—can process successfully.

Summary

A C program consists of one or more C functions. Every C program must contain a function called `main()` because it is the function called when the program starts up. A simple function consists of a function header followed by an opening brace, followed by the statements constituting the function body, followed by a terminating, or *closing*, brace.

Each C statement is an instruction to the computer and is marked by a terminating semicolon. A declaration statement creates a name for a variable and identifies the type of data to be stored in the variable. The name of a variable is an example of an identifier. An assignment statement assigns a value to a variable or, more generally, to a storage area. A function call statement causes the named function to be executed. When the called function is done, the program returns to the next statement after the function call.

The `printf()` function can be used to print phrases and the values of variables.

The *syntax* of a language is the set of rules that governs the way in which valid statements in that language are put together. The *semantics* of a statement is its meaning. The compiler helps you detect syntax errors, but semantic errors show up in a program's behavior only after it is compiled. Detecting semantic errors may involve tracing the program state—that is, the values of all variables—after each program step.

Finally, *keywords* are the vocabulary of the C language.

Review Questions

You'll find answers to the review questions in Appendix A, "Answers to the Review Questions."

1. What are the basic modules of a C program called?

2. What is a syntax error? Give an example of one in English and one in C.

3. What is a semantic error? Give an example of one in English and one in C.

4. Indiana Sloth has prepared the following program and brought it to you for approval. Please help him out.

```
include studio.h
int main{void} /* this prints the number of weeks in a year /*
(
int s

s := 56;
print(There are s weeks in a year.);
return 0;
```

5. Assuming that each of the following examples is part of a complete program, what will each one print?

```
a. printf("Baa Baa Black Sheep.");
   printf("Have you any wool?\n");
b. printf("Begone!\nO creature of lard!\n");
c. printf("What?\nNo/nfish?\n");
d. int num;

   num = 2;
   printf("%d + %d = %d", num, num, num + num);
```

6. Which of the following are C keywords? `main`, `int`, `function`, `char`, `=`

7. How would you print the values of the variables `words` and `lines` so they appear in the following form:

```
There were 3020 words and 350 lines.
```

   Here, `3020` and `350` represent the values of the two variables.

8. Consider the following program:

```
#include <stdio.h>
int main(void)
{
  int a, b;

     a = 5;
     b = 2;    /* line 7 */
     b = a;    /* line 8 */
     a = b;    /* line 9 */
     printf("%d %d\n", b, a);
     return 0;
}
```

   What is the program state after line 7? Line 8? Line 9?

9. Consider the following program:

```
#include <stdio.h>
int main(void)
{
    int x, y;

    x = 10;
    y = 5;        /* line 7 */
```

```
    y = x + y;    /* line 8 */
    x = x*y;      /* line 9 */
    printf("%d %d\n", x, y);
    return 0;
}
```

What is the program state after line 7? Line 8? Line 9?

Programming Exercises

Reading about C isn't enough. You should try writing one or two simple programs to see whether writing a program goes as smoothly as it looks in this chapter. A few suggestions follow, but you should also try to think up some problems yourself. You'll find answers to selected programming exercises on the publisher's website.

1. Write a program that uses one `printf()` call to print your first name and last name on one line, uses a second `printf()` call to print your first and last names on two separate lines, and uses a pair of `printf()` calls to print your first and last names on one line. The output should look like this (but using your name):

   ```
   Gustav Mahler ←First print statement
   Gustav        ←Second print statement
   Mahler        ←Still the second print statement
   Gustav Mahler ←Third and fourth print statements
   ```

2. Write a program to print your name and address.

3. Write a program that converts your age in years to days and displays both values. At this point, don't worry about fractional years and leap years.

4. Write a program that produces the following output:

   ```
   For he's a jolly good fellow!
   For he's a jolly good fellow!
   For he's a jolly good fellow!
   Which nobody can deny!
   ```

   Have the program use two user-defined functions in addition to `main()`: one named `jolly()` that prints the "jolly good" message once, and one named `deny()` that prints the final line once.

5. Write a program that produces the following output:

```
Brazil, Russia, India, China
India, China,
Brazil, Russia
```

   Have the program use two user-defined functions in addition to `main()`: one named `br()` that prints "Brazil, Russia" once, and one named `ic()` that prints "India, China" once. Let `main()` take care of any additional printing tasks.

6. Write a program that creates an integer variable called `toes`. Have the program set `toes` to `10`. Also have the program calculate what twice `toes` is and what `toes` squared is. The program should print all three values, identifying them.

7. Many studies suggest that smiling has benefits. Write a program that produces the following output:

```
Smile!Smile!Smile!
Smile!Smile!
Smile!
```

   Have the program define a function that displays the string `Smile!` once, and have the program use the function as often as needed.

8. In C, one function can call another. Write a program that calls a function named `one_three()`. This function should display the word `one` on one line, call a second function named `two()`, and then display the word `three` on one line. The function `two()` should display the word `two` on one line. The `main()` function should display the phrase `starting now:` before calling `one_three()` and display `done!` after calling it. Thus, the output should look like the following:

```
starting now:
one
two
three
done!
```