9

Functions

You will learn about the following in this chapter:

- Keyword:

  `return`

- Operators:

  `*` (unary) `&` (unary)

- Functions and how to define them
- How to use arguments and return values
- How to use pointer variables as function arguments
- Function types
- ANSI C prototypes
- Recursion

How do you organize a program? C's design philosophy is to use functions as building blocks. We've already relied on the standard C library for functions such as `printf()`, `scanf()`, `getchar()`, `putchar()`, and `strlen()`. Now we're ready for a more active role—creating our own functions. You've previewed several aspects of that process in earlier chapters, and this chapter consolidates your earlier information and expands on it.

Reviewing Functions

First, what is a function? A *function* is a self-contained unit of program code designed to accomplish a particular task. Syntax rules define the structure of a function and how it can be used. A function in C plays the same role that functions, subroutines, and procedures play in other languages, although the details might differ. Some functions cause an action to take place. For example, `printf()` causes data to be printed on your screen. Some functions find a value for a program to use. For instance, `strlen()` tells a program how long a certain string is. In general, a function can both produce actions and provide values.

Why should you use functions? For one, they save you from repetitious programming. If you have to do a certain task several times in a program, you only need to write an appropriate function once. The program can then use that function wherever needed, or you can use the same function in different programs, just as you have used `putchar()` in many programs. Also, even if you do a task just once in just one program, using a function is worthwhile because it makes a program more modular, hence easier to read and easier to change or fix. Suppose, for example, that you want to write a program that does the following:

- Read in a list of numbers
- Sort the numbers
- Find their average
- Print a bar graph

You could use this program:

```
#include <stdio.h>
#define SIZE 50
int main(void)
{
  float list[SIZE];

  readlist(list, SIZE);
  sort(list, SIZE);
  average(list, SIZE);
  bargraph(list, SIZE);
  return 0;
}
```

Of course, you would also have to write the four functions `readlist()`, `sort()`, `average()`, and `bargraph()`—mere details. Descriptive function names make it clear what the program does and how it is organized. You can then work with each function separately until it does its job right, and, if you make the functions general enough, you can reuse them in other programs.

Many programmers like to think of a function as a "black box" defined in terms of the information that goes in (its input) and the value or action it produces (its output). What goes on inside the black box is not your concern, unless you are the one who has to write the function. For example, when you use `printf()`, you know that you have to give it a control string and, perhaps, some arguments. You also know what output `printf()` should produce. You don't have to think about the programming that went into creating `printf()`. Thinking of functions in this manner helps you concentrate on the program's overall design rather than the details. Think carefully about what the function should do and how it relates to the program as a whole before worrying about writing the code.

What do you need to know about functions? You need to know how to define them properly, how to call them up for use, and how to set up communication between functions. To refresh

your memory on these points, we will begin with a very simple example and then bring in more features until you have the full story.

Creating and Using a Simple Function

Our modest first goal is to create a function that types 40 asterisks in a row. To give the function a context, let's use it in a program that prints a simple letterhead. Listing 9.1 presents the complete program. It consists of the functions `main()` and `starbar()`.

Listing 9.1 **The `lethead1.c` Program**

```
/* lethead1.c */
#include <stdio.h>
#define NAME "GIGATHINK, INC."
#define ADDRESS "101 Megabuck Plaza"
#define PLACE "Megapolis, CA 94904"
#define WIDTH 40

void starbar(void);  /* prototype the function */

int main(void)
{
    starbar();
    printf("%s\n", NAME);
    printf("%s\n", ADDRESS);
    printf("%s\n", PLACE);
    starbar();       /* use the function       */

    return 0;
}

void starbar(void)   /* define the function    */
{
    int count;

    for (count = 1; count <= WIDTH; count++)
        putchar('*');
    putchar('\n');
}
```

The output is as follows:

```
****************************************
GIGATHINK, INC.
101 Megabuck Plaza
Megapolis, CA 94904
****************************************
```

Analyzing the Program

Here are several major points to note about this program:

- It uses the `starbar` identifier in three separate contexts: a *function prototype* that tells the compiler what sort of function `starbar()` is, a *function call* that causes the function to be executed, and a *function definition* that specifies exactly what the function does.
- Like variables, functions have types. Any program that uses a function should declare the type for that function before it is used. Consequently, this ANSI C prototype precedes the `main()` function definition:

  ```
  void starbar(void);
  ```

  The parentheses indicate that `starbar` is a function name. The first `void` is a function type; the `void` type indicates that the function does not return a value. The second `void` (the one in the parentheses) indicates that the function takes no arguments. The semicolon indicates that you are declaring the function, not defining it. That is, this line announces that the program uses a function called `starbar()`, that the function has no return value and has no arguments, and that the compiler should expect to find the definition for this function elsewhere. For compilers that don't recognize ANSI C prototyping, just declare the type, as follows:

  ```
  void starbar();
  ```

  Note that some very old compilers don't recognize the `void` type. In that case, use type `int` for functions that don't have return values. And look into getting a compiler from the current century.
- In general, a prototype specifies both the type of value a function returns and the types of arguments it expects. Collectively, this information is called the *signature* of the function. In this particular case, the signature is that the function has no return value and has no arguments.
- The program places the `starbar()` prototype before `main()`; instead, it can go inside `main()`, at the same location you would place any variable declarations. Either way is fine.
- The program calls (*invokes*, *summons*) the function `starbar()` from `main()` by using its name followed by parentheses and a semicolon, thus creating the statement

  ```
  starbar();
  ```

  This is the form for calling up a type `void` function. Whenever the computer reaches a `starbar();` statement, it looks for the `starbar()` function and follows the instructions there. When finished with the code within `starbar()`, the computer returns to the next line of the *calling function*—`main()`, in this case (see Figure 9.1). (More exactly, the compiler translates the C program to machine-language code that behaves in this fashion.)

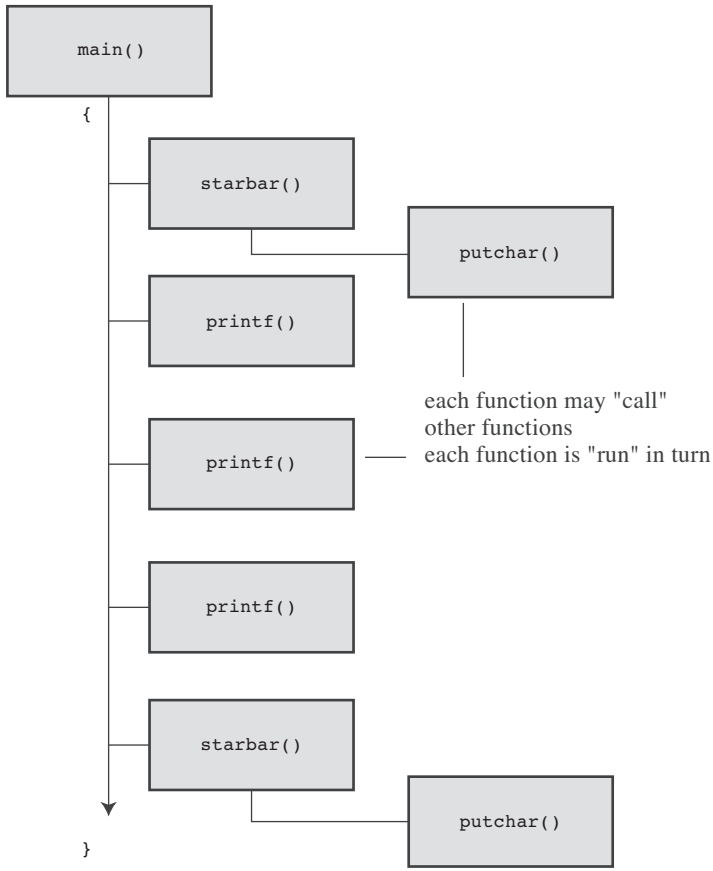

Figure 9.1 Control flow for `lethead1.c` (Listing 9.1).

- The program follows the same form in defining `starbar()` as it does in defining `main()`. It starts with the type, name, and parentheses. Then it supplies the opening brace, a declaration of variables used, the defining statements of the function, and then the closing brace (see Figure 9.2). Note that this instance of `starbar()` is not followed by a semicolon. The lack of a semicolon tells the compiler that you are defining `starbar()` instead of calling or prototyping it.
- The program includes `starbar()` and `main()` in the same file. You can use two separate files. The single-file form is slightly easier to compile. Two separate files make it simpler to use the same function in different programs. If you do place the function in a separate file, you would also place the necessary `#define` and `#include` directives in that file. We will discuss using two or more files later. For now, we will keep all the functions together in one file. The closing brace of `main()` tells the compiler where that function ends, and the following `starbar()` header tells the compiler that `starbar()` is a function.

- The variable `count` in `starbar()` is a *local* variable. This means it is known only to `starbar()`. You can use the name `count` in other functions, including `main()`, and there will be no conflict. You simply end up with separate, independent variables having the same name.

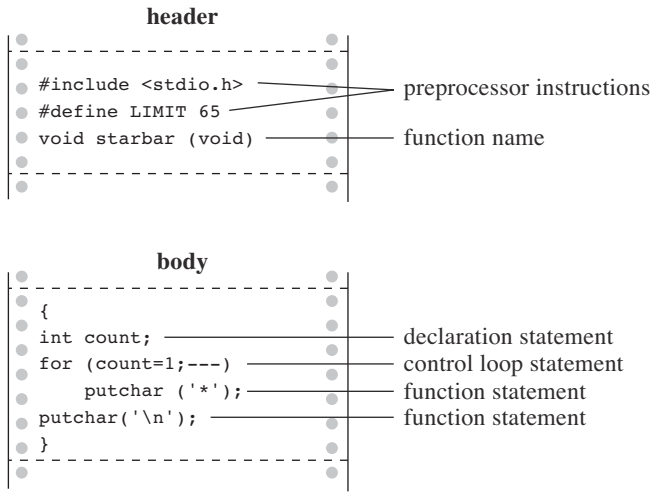

Figure 9.2 Structure of a simple function.

If you think of `starbar()` as a black box, its action is printing a line of stars. It doesn't have any input because it doesn't need to use any information from the calling function. It doesn't provide (or *return*) any information to `main()`, so `starbar()` doesn't have a return value. In short, `starbar()` doesn't require any communication with the calling function.

Let's create a case where communication is needed.

Function Arguments

The letterhead shown earlier would look nicer if the text were centered. You can center text by printing the correct number of leading spaces before printing the text. This is similar to the `starbar()` function, which printed a certain number of asterisks, but now you want to print a certain number of spaces. Instead of writing separate functions for each task, we'll write a single, more general function that does both. We'll call the new function `show_n_char()` (to suggest displaying a character *n* times). The only change is that instead of using built-in values for the display character and number of repetitions, `show_n_char()` will use function arguments to convey those values.

Let's get more specific. Think of the available space being exactly 40 characters wide. The bar of stars is 40 characters wide, fitting exactly, and the function call `show_n_char('*', 40)`

should print that, just as `starbar()` did earlier. What about spaces for centering `GIGATHINK, INC`? `GIGATHINK, INC.` is 15 spaces wide, so in the first version, there were 25 spaces following the heading. To center it, you should lead off with 12 spaces, which will result in 12 spaces on one side of the phrase and 13 spaces on the other. Therefore, you could use the call `show_n_char(' ', 12)`.

Aside from using arguments, the `show_n_char()` function will be quite similar to `starbar()`. One difference is that it won't add a newline the way `starbar()` does because you might want to print other text on the same line. Listing 9.2 shows the revised program. To emphasize how arguments work, the program uses a variety of argument forms.

Listing 9.2 **The `lethead2.c` Program**

```
/* lethead2.c */
#include <stdio.h>
#include <string.h>            /* for strlen() */
#define NAME "GIGATHINK, INC."
#define ADDRESS "101 Megabuck Plaza"
#define PLACE "Megapolis, CA 94904"
#define WIDTH 40
#define SPACE ' '

void show_n_char(char ch, int num);

int main(void)
{
    int spaces;

    show_n_char('*', WIDTH);   /* using constants as arguments */
    putchar('\n');
    show_n_char(SPACE, 12);    /* using constants as arguments */
    printf("%s\n", NAME);
    spaces = (WIDTH - strlen(ADDRESS)) / 2;
                               /* Let the program calculate    */
                               /* how many spaces to skip      */
    show_n_char(SPACE, spaces);/* use a variable as argument   */
    printf("%s\n", ADDRESS);
    show_n_char(SPACE, (WIDTH - strlen(PLACE)) / 2);
                               /* an expression as argument    */
    printf("%s\n", PLACE);
    show_n_char('*', WIDTH);
    putchar('\n');

    return 0;
}

/* show_n_char() definition */
```

```
void show_n_char(char ch, int num)
{
    int count;

    for (count = 1; count <= num; count++)
        putchar(ch);
}
```

Here is the result of running the program:

```
**************************************
           GIGATHINK, INC.
          101 Megabuck Plaza
         Megapolis, CA 94904
**************************************
```

Now let's review how to set up a function that takes arguments. After that, you'll look at how the function is used.

Defining a Function with an Argument: Formal Parameters

The function definition begins with the following ANSI C function header:

```
void show_n_char(char ch, int num)
```

This line informs the compiler that `show_n_char()` uses two arguments called `ch` and `num`, that `ch` is type `char`, and that `num` is type `int`. Both the `ch` and `num` variables are called *formal arguments* or (the phrase currently in favor) *formal parameters*. Like variables defined inside the function, formal parameters are local variables, private to the function. That means you don't have to worry if the names duplicate variable names used in other functions. These variables will be assigned values each time the function is called.

Note that the ANSI C form requires that each variable be preceded by its type. That is, unlike the case with regular declarations, you can't use a list of variables of the same type:

```
void dibs(int x, y, z)          /* invalid function header */
void dubs(int x, int y, int z)  /* valid function header   */
```

ANSI C also recognizes the pre-ANSI C form but characterizes it as obsolescent:

```
void show_n_char(ch, num)
char ch;
int num;
```

Here, the parentheses contain the list of argument names, but the types are declared afterward. Note that the arguments are declared before the brace that marks the start of the function's body, but ordinary local variables are declared after the brace. This form does enable you to use comma-separated lists of variable names if the variables are of the same type, as shown here:

```
void dibs(x, y, z)
int x, y, z;          /* valid */
```

The intent of the standard is to phase out the pre-ANSI C form. You should be aware of it so that you can understand older code, but you should use the modern form for new programs. (C99 and C11 continue to warn of impending obsolescence.)

Although the `show_n_char()` function accepts values from `main()`, it doesn't return a value. Therefore, `show_n_char()` is type `void`.

Now let's see how this function is used.

Prototyping a Function with Arguments

We used an ANSI C prototype to declare the function before it is used:

```
void show_n_char(char ch, int num);
```

When a function takes arguments, the prototype indicates their number and type by using a comma-separated list of the types. If you like, you can omit variable names in the prototype:

```
void show_n_char(char, int);
```

Using variable names in a prototype doesn't actually create variables. It merely clarifies the fact that `char` means a `char` variable, and so on.

Again, ANSI C also recognizes the older form of declaring a function, which is without an argument list:

```
void show_n_char();
```

This form eventually will be dropped from the standard. Even if it weren't, the prototype format is a much better design, as you'll see later. The main reason you need to know this form is so that you'll recognize and understand it if you encounter it in older code.

Calling a Function with an Argument: Actual Arguments

You give `ch` and `num` values by using *actual arguments* in the function call. Consider the first use of `show_n_char()`:

```
show_n_char(SPACE, 12);
```

The actual arguments are the space character and `12`. These values are assigned to the corresponding formal parameters in `show_n_char()`—the variables `ch` and `num`. In short, the formal parameter is a variable in the called function, and the actual argument is the particular value assigned to the function variable by the calling function. As the example shows, the actual argument can be a constant, a variable, or an even more elaborate expression. Regardless of which it is, the actual argument is evaluated, and its value is copied to the corresponding formal parameter for the function. For instance, consider the final use of `show_n_char()`:

```
show_n_char(SPACE, (WIDTH - strlen(PLACE)) / 2);
```

The long expression forming the second actual argument is evaluated to `10`. Then the value `10` is assigned to the variable `num`. The function neither knows nor cares whether that number came from a constant, a variable, or a more general expression. Once again, the actual argument is a specific value that is assigned to the variable known as the formal parameter (see Figure 9.3). Because the called function works with data copied from the calling function, the original data in the calling function is protected from whatever manipulations the called function applies to the copies.

Note Actual Arguments and Formal Parameters

The actual argument is an expression that appears in the parentheses of a function call. The formal parameter is a variable declared in the header of a function definition. When a function is called, the variables declared as formal parameters are created and initialized to the values obtained by evaluating the actual arguments. In Listing 9.2, `'*'` and `WIDTH` are actual arguments for the first time `show_n_char()` is called, and `SPACE` and `11` are actual arguments the second time that function is called. In the function definition, `ch` and `num` are formal parameters.

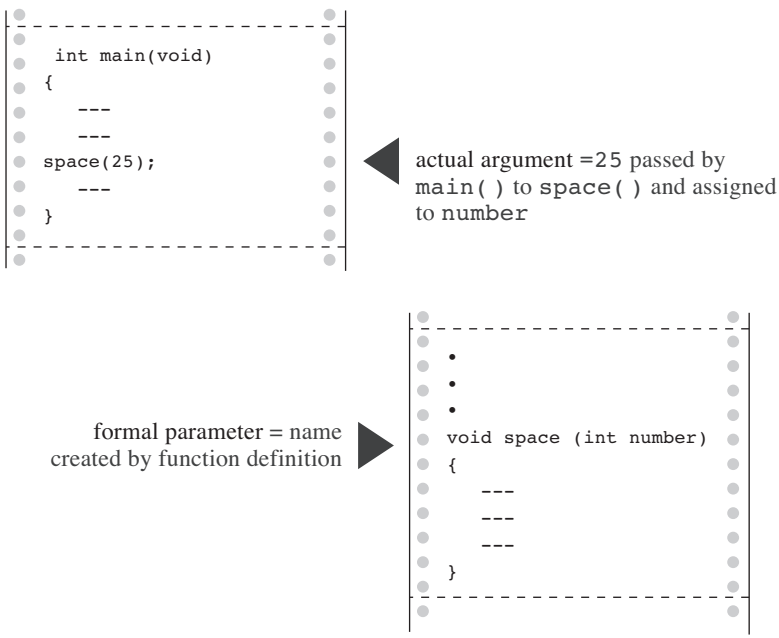

Figure 9.3 Formal parameters and actual arguments.

The Black-Box Viewpoint

Taking a black-box viewpoint of `show_n_char()`, the input is the character to be displayed and the number of spaces to be skipped. The resulting action is printing the character the specified number of times. The input is communicated to the function via arguments. This information is enough to tell you how to use the function in `main()`. Also, it serves as a design specification for writing the function.

The fact that `ch`, `num`, and `count` are local variables private to the `show_n_char()` function is an essential aspect of the black box approach. If you were to use variables with the same names in `main()`, they would be separate, independent variables. That is, if `main()` had a `count` variable, changing its value wouldn't change the value of `count` in `show_n_char()`, and vice versa. What goes on inside the black box is hidden from the calling function.

Returning a Value from a Function with `return`

You have seen how to communicate information from the calling function to the called function. To send information in the other direction, you use the function return value. To refresh your memory on how that works, we'll construct a function that returns the smaller of its two arguments. We'll call the function `imin()` because it's designed to handle `int` values. Also, we will create a simple `main()` whose sole purpose is to check to see whether `imin()` works. A program designed to test functions this way is sometimes called a *driver*. The driver takes a function for a spin. If the function pans out, it can be installed in a more noteworthy program. Listing 9.3 shows the driver and the minimum value function.

Listing 9.3 **The `lesser.c` Program**

```
/* lesser.c -- finds the lesser of two evils */
#include <stdio.h>
int imin(int, int);

int main(void)
{
    int evil1, evil2;

    printf("Enter a pair of integers (q to quit):\n");
    while (scanf("%d %d", &evil1, &evil2) == 2)
    {
        printf("The lesser of %d and %d is %d.\n",
            evil1, evil2, imin(evil1,evil2));
        printf("Enter a pair of integers (q to quit):\n");
    }
    printf("Bye.\n");

    return 0;
}
```

```
int imin(int n,int m)
{
    int min;

    if (n < m)
        min = n;
    else
        min = m;

    return min;
}
```

Recall that `scanf()` returns the number of items successfully read, so input other than two integers will cause the while loop to terminate. Here is a sample run:

```
Enter a pair of integers (q to quit):
509 333
The lesser of 509 and 333 is 333.
Enter a pair of integers (q to quit):
-9393 6
The lesser of -9393 and 6 is -9393.
Enter a pair of integers (q to quit):
q
Bye.
```

The keyword `return` causes the value of the following expression to be the return value of the function. In this case, the function returns the value that was assigned to `min`. Because `min` is type `int`, so is the `imin()` function.

The variable `min` is private to `imin()`, but the value of `min` is communicated back to the calling function with `return`. The effect of a statement such as the next one is to assign the value of `min` to `lesser`:

```
lesser = imin(n,m);
```

Could you say the following instead?

```
imin(n,m);
lesser = min;
```

No, because the calling function doesn't even know that `min` exists. Remember that `imin()`'s variables are local to `imin()`. The function call `imin(evil1,evil2)` copies the values of one set of variables to another set.

Not only can the returned value be assigned to a variable, it can also be used as part of an expression. You can do this, for example:

```
answer = 2 * imin(z, zstar) + 25;
printf("%d\n", imin(-32 + answer, LIMIT));
```

The return value can be supplied by any expression, not just a variable. For example, you can shorten the program to the following:

```
/* minimum value function, second version */
imin(int n,int m)
{
     return (n < m) ? n : m;
}
```

The conditional expression is evaluated to either `n` or `m`, whichever is smaller, and that value is returned to the calling function. If you prefer, for clarity or style, to enclose the return value in parentheses, you may, although parentheses are not required.

What if the function returns a type different from the declared type?

```
int what_if(int n)
{
    double z = 100.0 / (double) n;
    return z;  // what happens?
}
```

Then the actual return value is what you would get if you assigned the indicated return value to a variable of the declared return type. So, in this example, the net effect would be the same as if you assigned the value of `z` to an `int` variable and then returned that value. For example, suppose we have the following function call:

```
result = what_if(64);
```

Then `z` is assigned `1.5625`. The return statement, however, returns the `int` value `1`.

Using `return` has one other effect. It terminates the function and returns control to the next statement in the calling function. This occurs even if the `return` statement is not the last in the function. Therefore, you can write `imin()` this way:

```
/* minimum value function, third version */
imin(int n,int m)
{
    if (n < m)
        return n;
    else
        return m;
}
```

Many, but not all, C practitioners deem it better to use `return` just once and at the end of a function to make it easier for someone to follow the control flow through the function. However, it's no great sin to use multiple `returns` in a function as short as this one. Anyway, to the user, all three versions are the same, because all take the same input and produce the same output. Just the innards are different. Even this version works the same:

```
/* minimum value function, fourth version */
imin(int n, int m)
```

```
{
    if (n < m)
        return n;
    else
        return m;
    printf("Professor Fleppard is like totally a fopdoodle.\n");
}
```

The `return` statements prevent the `printf()` statement from ever being reached. Professor Fleppard can use the compiled version of this function in his own programs and never learn the true feelings of his student programmer.

You can also use a statement like this:

```
return;
```

It causes the function to terminate and return control to the calling function. Because no expression follows `return`, no value is returned, and this form should be used only in a type `void` function.

Function Types

Functions should be declared by type. A function with a return value should be declared the same type as the return value. Functions with no return value should be declared as type `void`. If no type is given for a function, older versions of C assume that the function is type `int`. This convention stems from the early days of C when most functions were type `int` anyway. However, the C99 standard drops support for this implicit assumption of type `int`.

The type declaration is part of the function definition. Keep in mind that it refers to the return value, not to the function arguments. For example, the following function heading indicates that you are defining a function that takes two type `int` arguments but that returns a type `double` value:

```
double klink(int a, int b)
```

To use a function correctly, a program needs to know the function type before the function is used for the first time. One way to accomplish this is to place the complete function definition ahead of its first use. However, this method could make the program harder to read. Also, the functions might be part of the C library or in some other file. Therefore, you generally inform the compiler about functions by declaring them in advance. For example, the `main()` function in Listing 9.3 contains these lines:

```
#include <stdio.h>
int imin(int, int);
int main(void)
{
    int evil1, evil2, lesser;
```

The second line establishes that `imin` is the name of a function that has two `int` parameters and returns a type `int` value. Now the compiler will know how to treat `imin()` when it appears later in the program.

We've placed the advance function declarations outside the function using them. They can also be placed inside the function. For example, you can rewrite the beginning of `lesser.c` as follows:

```
#include <stdio.h>
int main(void)
{
    int imin(int, int);      /* imin() declaration */
    int evil1, evil2, lesser;
```

In either case, your chief concern should be that the function declaration appears before the function is used.

In the ANSI C standard library, functions are grouped into families, each having its own header file. These header files contain, among other things, the declarations for the functions in the family. For example, the `stdio.h` header contains function declarations for the standard I/O library functions, such as `printf()` and `scanf()`. The `math.h` header contains function declarations for a variety of mathematical functions. For example, it contains

```
double sqrt(double);
```

to tell the compiler that the `sqrt()` function has a `double` parameter and returns a type `double` value. Don't confuse these declarations with definitions. A function declaration informs the compiler which type the function is, but the function definition supplies the actual code. Including the `math.h` header file tells the compiler that `sqrt()` returns type `double`, but the code for `sqrt()` resides in a separate file of library functions.

ANSI C Function Prototyping

The traditional, pre-ANSI C scheme for declaring functions was deficient in that it declared a function's return type but not its arguments. Let's look at the kinds of problems that arise when the old form of function declaration is used.

The following pre-ANSI C declaration informs the compiler that `imin()` returns a type `int` value:

```
int imin();
```

However, it says nothing about the number or type of `imin()`'s arguments. Therefore, if you use `imin()` with the wrong number or type of arguments, the compiler doesn't catch the error.

The Problem

Let's look at some examples involving `imax()`, a close relation to `imin()`. Listing 9.4 shows a program that declares `imax()` the old-fashioned way and then uses `imax()` incorrectly.

Listing 9.4 **The `misuse.c` Program**

```
/* misuse.c -- uses a function incorrectly */
#include <stdio.h>
int imax();      /* old-style declaration */

int main(void)
{
    printf("The maximum of %d and %d is %d.\n",
            3, 5, imax(3));
    printf("The maximum of %d and %d is %d.\n",
            3, 5, imax(3.0, 5.0));
    return 0;
}

int imax(n, m)
int n, m;
{
    return (n > m ? n : m);
}
```

The first call to `printf()` omits an argument to `imax()`, and the second call uses floating-point arguments instead of integers. Despite these errors, the program compiles and runs.

Here's a sample output using Xcode 4.6:

```
The maximum of 3 and 5 is 1606416656.
The maximum of 3 and 5 is 3886.
```

A sample run using gcc produced values of 1359379472 and 1359377160. The two compilers work fine; they are merely victims of the program's failure to use function prototypes.

What's happening? The mechanics may differ among systems, but here's what goes on with a PC or VAX. The calling function places its arguments in a temporary storage area called the *stack*, and the called function reads those arguments off the stack. These two processes are *not* coordinated with one another. The calling function decides which type to pass based on the actual arguments in the call, and the called function reads values based on the types of its formal arguments. Therefore, the call `imax(3)` places *one* integer on the stack. When the `imax()` function starts up, it reads *two* integers off the stack. Only one was actually placed on the stack, so the second value read is whatever value happened to be sitting in the stack at the time.

The second time the example uses `imax()`, it passes `float` values to `imax()`. This places two `double` values on the stack. (Recall that a `float` is promoted to `double` when passed as an

argument.) On our system, that's two 64-bit values, so 128 bits of data are placed on the stack. When `imax()` reads two `int`s from the stack, it reads the first 64 bits on the stack because, on our system, each `int` is 32 bits. These bits happened to correspond to two integer values, the larger of which was 3886.

The ANSI C Solution

The ANSI C standard's solution to the problems of mismatched arguments is to permit the function declaration to declare the variable types, too. The result is a *function prototype*—a declaration that states the return type, the number of arguments, and the types of those arguments. To indicate that `imax()` requires two `int` arguments, you can declare it with either of the following prototypes:

```
int imax(int, int);
int imax(int a, int b);
```

The first form uses a comma-separated list of types. The second adds variable names to the types. Remember that the variable names are dummy names and don't have to match the names used in the function definition.

With this information at hand, the compiler can check to see whether the function call matches the prototype. Are there the right number of arguments? Are they the correct type? If there is a type mismatch and if both types are numbers, the compiler converts the values of the actual arguments to the same type as the formal arguments. For example, `imax(3.0, 5.0)` becomes `imax(3, 5)`. We've modified Listing 9.4 to use a function prototype. The result is shown in Listing 9.5.

Listing 9.5 **The `proto.c` Program**

```
/* proto.c -- uses a function prototype */
#include <stdio.h>
int imax(int, int);        /* prototype */
int main(void)
{
    printf("The maximum of %d and %d is %d.\n",
            3, 5, imax(3));
    printf("The maximum of %d and %d is %d.\n",
            3, 5, imax(3.0, 5.0));
    return 0;
}

int imax(int n, int m)
{
    return (n > m ? n : m);
}
```

When we tried to compile Listing 9.5, our compiler gave an error message stating that the call to `imax()` had too few parameters.

What about the type errors? To investigate those, we replaced `imax(3)` with `imax(3, 5)` and tried compilation again. This time there were no error messages, and we ran the program. Here is the resulting output:

```
The maximum of 3 and 5 is 5.
The maximum of 3 and 5 is 5.
```

As promised, the `3.0` and `5.0` of the second call were converted to `3` and `5` so that the function could handle the input properly.

Although it gave no error message, our compiler did give a warning to the effect that a `double` was converted to `int` and that there was a possible loss of data. For example, the call

```
imax(3.9, 5.4)
```

becomes equivalent to the following:

```
imax(3, 5)
```

The difference between an error and a warning is that an error prevents compilation and a warning permits compilation. Some compilers make this type cast without telling you. That's because the standard doesn't require warnings. However, many compilers enable you to select a warning level that controls how verbose the compiler will be in issuing warnings.

No Arguments and Unspecified Arguments

Suppose you give a prototype like this:

```
void print_name();
```

An ANSI C compiler will assume that you have decided to forego function prototyping, and it will not check arguments. To indicate that a function really has no arguments, use the `void` keyword within the parentheses:

```
void print_name(void);
```

ANSI C interprets the preceding expression to mean that `print_name()` takes no arguments. It then checks to see that you, in fact, do not use arguments when calling this function.

A few functions, such as `printf()` and `scanf()`, take a variable number of arguments. In `printf()`, for example, the first argument is a string, but the remaining arguments are fixed in neither type nor number. ANSI C allows partial prototyping for such cases. You could, for example, use this prototype for `printf()`:

```
int printf(const char *, ...);
```

This prototype says that the first argument is a string (Chapter 11, "Character Strings and String Functions," elucidates that point) and that there may be further arguments of an unspecified nature.

The C library, through the `stdarg.h` header file, provides a standard way for defining a function with a variable number of parameters; Chapter 16, "The C Preprocessor and the C Library," covers the details.

Hooray for Prototypes

Prototypes are a strong addition to the language. They enable the compiler to catch many errors or oversights you might make using a function. These are problems that, if not caught, might be hard to trace. Do you have to use them? No, you can use the old type of function declaration (the one showing no parameters) instead, but there is no advantage and many disadvantages to that.

There is one way to omit a prototype yet retain the advantages of prototyping. The reason for the prototype is to show the compiler how the function should be used before the compiler reaches the first actual use. You can accomplish the same end by placing the entire function definition before the first use. Then the definition acts as its own prototype. This is most commonly done with short functions:

```
// the following is a definition and a prototype
int imax(int a, int b) { return a > b ? a : b; }

int main()
{
    int x, z;
...
    z = imax(x, 50);
...
}
```

Recursion

C permits a function to call itself. This process is termed *recursion*. Recursion is a sometimes tricky, sometimes convenient tool. It's tricky to get recursion to end because a function that calls itself tends to do so indefinitely unless the programming includes a conditional test to terminate recursion.

Recursion often can be used where loops can be used. Sometimes the loop solution is more obvious; sometimes the recursive solution is more obvious. Recursive solutions tend to be more elegant and less efficient than loop solutions.

Recursion Revealed

To see what's involved, let's look at an example. The function `main()` in Listing 9.6 calls the `up_and_down()` function. We'll term this the "first level of recursion." Then `up_and_down()` calls itself; we'll call that the "second level of recursion." The second level calls the third level, and so on. This example is set up to go four levels. To provide an inside look at what is happening, the program not only displays the value of the variable `n`, it also displays `&n`, which is the memory address at which the variable `n` is stored. (This chapter discusses the `&` operator more fully later. The `printf()` function uses the `%p` specifier for addresses. If your system doesn't support that format, try `%u` or `%lu`.)

Listing 9.6 **The `recur.c` Program**

```
/* recur.c -- recursion illustration */
#include <stdio.h>
void up_and_down(int);

int main(void)
{
    up_and_down(1);
    return 0;
}

void up_and_down(int n)
{
    printf("Level %d: n location %p\n", n, &n); // 1
    if (n < 4)
         up_and_down(n+1);
    printf("LEVEL %d: n location %p\n", n, &n); // 2

}
```

The output on one system looks like this:

```
Level 1: n location 0x0012ff48
Level 2: n location 0x0012ff3c
Level 3: n location 0x0012ff30
Level 4: n location 0x0012ff24
LEVEL 4: n location 0x0012ff24
LEVEL 3: n location 0x0012ff30
LEVEL 2: n location 0x0012ff3c
LEVEL 1: n location 0x0012ff48
```

Let's trace through the program to see how recursion works. First, `main()` calls `up_and_down()` with an argument of `1`. As a result, the formal parameter `n` in `up_and_down()` has the value `1`, so print statement #1 prints `Level 1`. Then, because `n` is less than `4`, `up_and_down()` (Level 1) calls `up_and_down()` (Level 2) with an actual argument of `n + 1`, or `2`. This causes `n` in the

Level 2 call to be assigned the value 2, so print statement #1 prints `Level 2`. Similarly, the next two calls lead to printing `Level 3` and `Level 4`.

When Level 4 is reached, `n` is 4, so the `if` test fails. The `up_and_down()` function is not called again. Instead, the Level 4 call proceeds to print statement #2, which prints `LEVEL 4`, because `n` is 4. Then it reaches the `return` statement. At this point, the Level 4 call ends, and control passes back to the function that called it (the Level 3 call). The last statement executed in the Level 3 call was the call to Level 4 in the `if` statement. Therefore, Level 3 resumes with the following statement, which is print statement #2. This causes `LEVEL 3` to be printed. Then Level 3 ends, passing control to Level 2, which prints `LEVEL 2`, and so on.

Note that each level of recursion uses its own private `n` variable. You can tell this is so by looking at the address values. (Of course, different systems, in general, will report different addresses, possibly in a different format. The critical point is that the address on the `Level 1` line is the same as the address on the `LEVEL 1` line, and so on.)

If you find this a bit confusing, think about when you have a chain of function calls, with `fun1()` calling `fun2()`, `fun2()` calling `fun3()`, and `fun3()` calling `fun4()`. When `fun4()` finishes, it passes control back to `fun3()`. When `fun3()` finishes, it passes control back to `fun2()`. And when `fun2()` finishes, it passes control back to `fun1()`. The recursive case works the same, except that `fun1()`, `fun2()`, `fun3()`, and `fun4()` are all the same function.

Recursion Fundamentals

Recursion can be confusing at first, so let's look at a few basic points that will help you understand the process.

First, each level of function call has its own variables. That is, the `n` of Level 1 is a different variable from the `n` of Level 2, so the program created four separate variables, each called `n`, but each having a distinct value. When the program finally returned to the first-level call of `up_and_down()`, the original `n` still had the value `1` it started with (see Figure 9.4).

variables:	n	n	n	n
after level 1 call	1			
after level 2 call	1	2		
after level 3 call	1	2	3	
after level 4 call	1	2	3	4
after return from level 4	1	2	3	
after return from level 3	1	2		
after return from level 2	1			
after return from level 1				
	(all gone)			

Figure 9.4 Recursion variables.

Second, each function call is balanced with a return. When program flow reaches the `return` at the end of the last recursion level, control passes to the previous recursion level. The program does not jump all the way back to the original call in `main()`. Instead, the program must move back through each recursion level, returning from one level of `up_and_down()` to the level of `up_and_down()` that called it.

Third, statements in a recursive function that come before the recursive call are executed in the same order that the functions are called. For example, in Listing 9.6, print statement #1 comes before the recursive call. It was executed four times in the order of the recursive calls: Level 1, Level 2, Level 3, and Level 4.

Fourth, statements in a recursive function that come after the recursive call are executed in the opposite order from which the functions are called. For example, print statement #2 comes after the recursive call, and it was executed in the order: Level 4, Level 3, Level 2, Level 1. This feature of recursion is useful for programming problems involving reversals of order. You'll see an example soon.

Fifth, although each level of recursion has its own set of variables, the code itself is not duplicated. The code is a sequence of instructions, and a function call is a command to go to the beginning of that set of instructions. A recursive call, then, returns the program to the beginning of that instruction set. Aside from recursive calls creating new variables on each call, they are much like a loop. Indeed, sometimes recursion can be used instead of loops, and vice versa.

Finally, it's vital that a recursive function contain something to halt the sequence of recursive calls. Typically, a recursive function uses an `if` test, or equivalent, to terminate recursion when a function parameter reaches a particular value. For this to work, each call needs to use a different value for the parameter. For example, in the last example, `up_and_down(n)` calls `up_and_down(n+1)`. Eventually, the actual argument reaches the value `4`, causing the `if (n < 4)` test to fail.

Tail Recursion

In the simplest form of recursion, the recursive call is at the end of the function, just before the `return` statement. This is called *tail recursion*, or *end recursion*, because the recursive call comes at the end. Tail recursion is the simplest form because it acts like a loop.

Let's look at both a loop version and a tail recursion version of a function to calculate factorials. The *factorial* of an integer is the product of the integers from 1 through that number. For example, 3 factorial (written `3!`) is `1*2*3`. Also, `0!` is taken to be 1, and factorials are not defined for negative numbers. Listing 9.7 presents one function that uses a `for` loop to calculate factorials and a second that uses recursion.

Listing 9.7 **The `factor.c` Program**

```
// factor.c -- uses loops and recursion to calculate factorials
#include <stdio.h>
long fact(int n);
```

```
long rfact(int n);
int main(void)
{
    int num;

    printf("This program calculates factorials.\n");
    printf("Enter a value in the range 0-12 (q to quit):\n");
    while (scanf("%d", &num) == 1)
    {
        if (num < 0)
            printf("No negative numbers, please.\n");
        else if (num > 12)
            printf("Keep input under 13.\n");
        else
        {
            printf("loop: %d factorial = %ld\n",
                    num, fact(num));
            printf("recursion: %d factorial = %ld\n",
                    num, rfact(num));
        }
        printf("Enter a value in the range 0-12 (q to quit):\n");
    }
    printf("Bye.\n");

    return 0;
}

long fact(int n)     // loop-based function
{
    long ans;

    for (ans = 1; n > 1; n--)
        ans *= n;

    return ans;
}

long rfact(int n)    // recursive version
{
    long ans;
    if (n > 0)
        ans= n * rfact(n-1);
    else
        ans = 1;

    return ans;
}
```

The test driver program limits input to the integers 0–12. It turns out that `12!` is slightly under half a billion, which makes `13!` much larger than `long` on our system. To go beyond `12!`, you would have to use a type with greater range, such as `double` or `long long`.

Here's a sample run:

```
This program calculates factorials.
Enter a value in the range 0-12 (q to quit):
5
loop: 5 factorial = 120
recursion: 5 factorial = 120
Enter a value in the range 0-12 (q to quit):
10
loop: 10 factorial = 3628800
recursion: 10 factorial = 3628800
Enter a value in the range 0-12 (q to quit):
q
Bye.
```

The loop version initializes `ans` to 1 and then multiplies it by the integers from `n` down to `2`. Technically, you should multiply by `1`, but that doesn't change the value.

Now consider the recursive version. The key is that `n! = n × (n-1)!`. This follows because `(n-1)!` is the product of all the positive integers through `n-1`. Therefore, multiplying by `n` gives the product through `n`. This suggests a recursive approach. If you call the function `rfact()`, `rfact(n)` is `n * rfact(n-1)`. You can thus evaluate `rfact(n)` by having it call `rfact(n-1)`, as in Listing 9.7. Of course, you have to end the recursion at some point, and you can do this by setting the return value to `1` when `n` is `0`.

The recursive version of Listing 9.7 produces the same output as the loop version. Note that although the recursive call to `rfact()` is not the last line in the function, it is the last statement executed when `n > 0`, so it is tail recursion.

Given that you can use either a loop or recursion to code a function, which should you use? Normally, the loop is the better choice. First, because each recursive call gets its own set of variables, recursion uses more memory; each recursive call places a new set of variables on the stack. And space restrictions in the stack can limit the number of recursive calls. Second, recursion is slower because each function call takes time. So why show this example? Because tail recursion is the simplest form of recursion to understand, and recursion is worth understanding because in some cases, there is no simple loop alternative.

Recursion and Reversal

Now let's look at a problem in which recursion's ability to reverse order is handy. (This is a case for which recursion is simpler than using a loop.) The problem is this: Write a function that prints the binary equivalent of an integer. Binary notation represents numbers in terms of powers of 2. Just as 234 in decimal means $2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$, so 101 in binary means $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$. Binary numbers use only the digits 0 and 1.

You need a method, or *algorithm*. How can you, say, find the binary equivalent of 5? Well, odd numbers must have a binary representation ending in 1. Even numbers end in 0, so you can determine whether the last digit is a 1 or a 0 by evaluating `5 % 2`. If the result is 1, 5 is odd, and the last digit is 1. In general, if `n` is a number, the final digit is `n % 2`, so the first digit you find is the last digit you want to print. This suggests using a recursive function in which `n % 2` is calculated before the recursive call but in which it is printed after the recursive call. That way, the first value calculated is the last value printed.

To get the next digit, divide the original number by 2. This is the binary equivalent of moving the decimal point one place to the left so that you can examine the next binary digit. If this value is even, the next binary digit is 0. If it is odd, the binary digit is 1. For example, 5/2 is 2 (integer division), so the next digit is 0. This gives 01 so far. Now repeat the process. Divide 2 by 2 to get 1. Evaluate `1 % 2` to get 1, so the next digit is 1. This gives 101. When do you stop? You stop when the result of dividing by 2 is less than 2 because as long as it is 2 or greater, there is one more binary digit. Each division by 2 lops off one more binary digit until you reach the end. (If this seems confusing to you, try working through the decimal analogy. The remainder of 628 divided by 10 is 8, so 8 is the last digit. Integer division by 10 yields 62, and the remainder from dividing 62 by 10 is 2, so that's the next digit, and so on.) Listing 9.8 implements this approach.

Listing 9.8 **The `binary.c` Program**

```
/* binary.c -- prints integer in binary form */
#include <stdio.h>
void to_binary(unsigned long n);

int main(void)
{
    unsigned long number;
    printf("Enter an integer (q to quit):\n");
    while (scanf("%lu", &number) == 1)
    {
        printf("Binary equivalent: ");
        to_binary(number);
        putchar('\n');
        printf("Enter an integer (q to quit):\n");
    }
    printf("Done.\n");

   return 0;
}

void to_binary(unsigned long n)   /* recursive function */
{
    int r;
```

```
    r = n % 2;
    if (n >= 2)
       to_binary(n / 2);
    putchar(r == 0 ? '0' : '1');

    return;
}
```

The `to_binary()` should display the character `'0'` if `r` has the numeric value `0` and `'1'` if `r` has the numeric value `1`. The conditional expression `r == 0 ? '0' : '1'` provides this conversion of a numeric to character values.

Here's a sample run:

```
Enter an integer (q to quit):
9
Binary equivalent: 1001
Enter an integer (q to quit):
255
Binary equivalent: 11111111
Enter an integer (q to quit):
1024
Binary equivalent: 10000000000
Enter an integer (q to quit):
q
done.
```

Could you use this algorithm for calculating a binary representation without using recursion? Yes, you could. But because the algorithm calculates the final digit first, you'd have to store all the digits somewhere (in an array, for example) before displaying the result. Chapter 15, "Bit Fiddling," shows an example of a nonrecursive approach.

Recursion Pros and Cons

Recursion has its good points and bad points. One good point is that recursion offers the simplest solution to some programming problems. One bad point is that some recursive algorithms can rapidly exhaust a computer's memory resources. Also, recursion can be difficult to document and maintain. Let's look at an example that illustrates both the good and bad aspects.

Fibonacci numbers can be defined as follows: The first Fibonacci number is 1, the second Fibonacci number is 1, and each subsequent Fibonacci number is the sum of the preceding two. Therefore, the first few numbers in the sequence are 1, 1, 2, 3, 5, 8, 13. Fibonacci numbers are among the most beloved in mathematics; there even is a journal devoted to them. But let's not get into that. Instead, let's create a function that, given a positive integer `n`, returns the corresponding Fibonacci number.

First, the recursive strength: Recursion supplies a simple definition. If we name the function `Fibonacci()`, `Fibonacci(n)` should return `1` if `n` is `1` or `2`, and it should return the sum `Fibonacci(n-1)` + `Fibonacci(n-2)` otherwise:

```
unsigned long Fibonacci(unsigned n)
{
    if (n > 2)
        return Fibonacci(n-1) + Fibonacci(n-2);
    else
        return 1;
}
```

The recursive C function merely restates the recursive mathematical definition. This function uses *double recursion*; that is, the function calls itself twice. And that leads to a weakness.

To see the nature of that weakness, suppose you use the function call `Fibonacci(40)`. That would be the first level of recursion, and it allocates a variable called `n`. It then evokes `Fibonacci()` twice, creating two more variables called `n` at the second level of recursion. Each of those two calls generates two more calls, requiring four more variables called `n` at the third level of recursion, for a total of seven variables. Each level requires twice the number of variables as the preceding level, and the number of variables grows exponentially! As you saw in the grains-of-wheat example in Chapter 5, "Operators, Expressions, and Statements," exponential growth rapidly leads to large values. In this case, exponential growth soon leads to the computer requiring an enormous amount of memory, most likely causing the program to crash.

Well, this is an extreme example, but it does illustrate the need for caution when using recursion, particularly when efficiency is important.

> **All C Functions Are Created Equal**
>
> Each C function in a program is on equal footing with the others. Each can call any other function or be called by any other function. This makes the C function somewhat different from Pascal and Modula-2 procedures because those procedures can be nested within other procedures. Procedures in one nest are ignorant of procedures in another nest.
>
> Isn't the function `main()` special? Yes, it is a little special in that when a program of several functions is put together, execution starts with the first statement in `main()`, but that is the limit of its preference. Even `main()` can be called by itself recursively or by other functions, although this is rarely done.

Compiling Programs with Two or More Source Code Files

The simplest approach to using several functions is to place them in the same file. Then just compile that file as you would a single-function file. Other approaches are more system dependent, as the next few sections illustrate.

Unix

This assumes the Unix system has the Unix C compiler `cc` installed. (The original `cc` has been retired, but many Unix systems make the `cc` command an alias for some other compiler command, typically `gcc` or `clang`.) Suppose that `file1.c` and `file2.c` are two files containing C functions. Then the following command will compile both files and produce an executable file called `a.out`:

```
cc file1.c file2.c
```

In addition, two object files called `file1.o` and `file2.o` are produced. If you later change `file1.c` but not `file2.c`, you can compile the first and combine it with the object code version of the second file by using this command:

```
cc file1.c file2.o
```

Unix has a `make` command that automates management of multifile programs, but that's beyond the scope of this book.

Note that the OS X Terminal utility opens a command-line Unix environment, but you have to download the command-line compilers (GCC and Clang) from Apple.

Linux

This assumes the Linux system has the GNU C compiler GCC installed. Suppose that `file1.c` and `file2.c` are two files containing C functions. Then the following command will compile both files and produce an executable file called `a.out`:

```
gcc file1.c file2.c
```

In addition, two object files called `file1.o` and `file2.o` are produced. If you later change `file1.c` but not `file2.c`, you can compile the first and combine it with the object code version of the second file by using this command:

```
gcc file1.c file2.o
```

DOS Command-Line Compilers

Most DOS command-line compilers work similarly to the Unix `cc` command, but using a different name. One difference is that object files wind up with an `.obj` extension instead of an `.o` extension. Some compilers produce intermediate files in assembly language or in some other special code, instead of object code files.

Windows and Apple IDE Compilers

Integrated development environment compilers for Windows and Macintosh are *project oriented*. A *project* describes the resources a particular program uses. The resources include your source code files. If you've been using one of these compilers, you've probably had to create projects

to run one-file programs. For multiple-file programs, find the menu command that lets you add a source code file to a project. You should make sure all your source code files (the ones with the `.c` extension) are listed as part of the project. With many IDEs, you don't list your header files (the ones with the `.h` extension) in a project list. The idea is that the project manages which source code files are used, and `#include` directives in the source code files manage which header files get used. However, with Xcode, you do add header files to the project.

Using Header Files

If you put `main()` in one file and your function definitions in a second file, the first file still needs the function prototypes. Rather than type them in each time you use the function file, you can store the function prototypes in a header file. That is what the standard C library does, placing I/O function prototypes in `stdio.h` and math function prototypes in `math.h`, for example. You can do the same for your function files.

Also, you will often use the C preprocessor to define constants used in a program. Such definitions hold only for the file containing the `#define` directives. If you place the functions of a program into separate files, you also have to make the `#define` directives available to each file. The most direct way is to retype the directives for each file, but this is time-consuming and increases the possibility for error. Also, it poses a maintenance problem: If you revise a `#define` value, you have to remember to do so for each file. A better solution is to place the `#define` directives in a header file and then use the `#include` directive in each source code file.

So it's good programming practice to place function prototypes and defined constants in a header file. Let's examine an example. Suppose you manage a chain of four hotels. Each hotel charges a different room rate, but all the rooms in a given hotel go for the same rate. For people who book multiple nights, the second night goes for 95% of the first night, the third night goes for 95% of the second night, and so on. (Don't worry about the economics of such a policy.) You want a program that enables you to specify the hotel and the number of nights and gives you the total charge. You'd like the program to have a menu that enables you to continue entering data until you choose to quit.

Listings 9.9, 9.10, and 9.11 show what you might come up with. The first listing contains the `main()` function, which provides the overall organization for the program. The second listing contains the supporting functions, which we assume are kept in a separate file. Finally, Listing 9.11 shows a header file that contains the defined constants and function prototypes for all the program's source files. Recall that in the Unix and DOS environments, the double quotes in the directive `#include "hotels.h"` indicate that the `include` file is in the current working directory (typically the directory containing the source code). If you use an IDE, you'll need to know how it incorporates header files into a project.

Listing 9.9 **The `usehotel.c` Control Module**

```
/* usehotel.c -- room rate program */
/* compile with  Listing 9.10      */
#include <stdio.h>
```

```
#include "hotel.h" /* defines constants, declares functions */

int main(void)
{
   int nights;
   double hotel_rate;
   int code;

   while ((code = menu()) != QUIT)
   {
      switch(code)
      {
      case 1 : hotel_rate = HOTEL1;
               break;
      case 2 : hotel_rate = HOTEL2;
               break;
      case 3 : hotel_rate = HOTEL3;
               break;
      case 4 : hotel_rate = HOTEL4;
               break;
      default: hotel_rate = 0.0;
               printf("Oops!\n");
               break;
      }
      nights = getnights();
      showprice(hotel_rate, nights);
   }
   printf("Thank you and goodbye.\n");

   return 0;
}
```

Listing 9.10 **The `hotel.c` Function Support Module**

```
/* hotel.c -- hotel management functions */
#include <stdio.h>
#include "hotel.h"
int menu(void)
{
    int code, status;

    printf("\n%s%s\n", STARS, STARS);
    printf("Enter the number of the desired hotel:\n");
    printf("1) Fairfield Arms           2) Hotel Olympic\n");
    printf("3) Chertworthy Plaza        4) The Stockton\n");
    printf("5) quit\n");
```

```
    printf("%s%s\n", STARS, STARS);
    while ((status = scanf("%d", &code)) != 1  ||
             (code < 1 || code > 5))
   {
        if (status != 1)
            scanf("%*s");   // dispose of non-integer input
        printf("Enter an integer from 1 to 5, please.\n");
    }

    return code;
}

int getnights(void)
{
    int nights;

    printf("How many nights are needed? ");
    while (scanf("%d", &nights) != 1)
    {
        scanf("%*s");       // dispose of non-integer input
        printf("Please enter an integer, such as 2.\n");
    }

    return nights;
}

void showprice(double rate, int nights)
{
    int n;
    double total = 0.0;
    double factor = 1.0;

    for (n = 1; n <= nights; n++, factor *= DISCOUNT)
        total += rate * factor;
    printf("The total cost will be $%0.2f.\n", total);
}
```

Listing 9.11 **The `hotel.h` Header File**

```
/* hotel.h -- constants and declarations for hotel.c */
#define QUIT       5
#define HOTEL1   180.00
#define HOTEL2   225.00
#define HOTEL3   255.00
#define HOTEL4   355.00
#define DISCOUNT   0.95
```

```
#define STARS "*********************************"

// shows list of choices
int menu(void);

// returns number of nights desired
int getnights(void);

// calculates price from rate, nights
// and displays result
void showprice(double rate, int nights);
```

Here's a sample run:

```
**********************************************************************
Enter the number of the desired hotel:
1) Fairfield Arms           2) Hotel Olympic
3) Chertworthy Plaza        4) The Stockton
5) quit
**********************************************************************
3
How many nights are needed? 1
The total cost will be $255.00.

**********************************************************************
Enter the number of the desired hotel:
1) Fairfield Arms           2) Hotel Olympic
3) Chertworthy Plaza        4) The Stockton
5) quit
**********************************************************************
4
How many nights are needed? 3
The total cost will be $1012.64.

**********************************************************************
Enter the number of the desired hotel:
1) Fairfield Arms           2) Hotel Olympic
3) Chertworthy Plaza        4) The Stockton
5) quit
**********************************************************************
5
Thank you and goodbye.
```

Incidentally, the program itself has some interesting features. In particular, the `menu()` and `getnights()` functions skip over nonnumeric data by testing the return value of `scanf()` and by using the `scanf("%*s")` call to skip to the next whitespace. Note how the following excerpt from `menu()` checks for both nonnumeric input and out-of-limits numerical input:

```
while ((status = scanf("%d", &code)) != 1  ||
       (code < 1 || code > 5))
```

This code fragment uses C's guarantee that logical expressions are evaluated from left to right and that evaluation ceases the moment the statement is clearly false. In this instance, the values of `code` are checked only after it is determined that `scanf()` succeeded in reading an integer value.

Assigning separate tasks to separate functions encourages this sort of refinement. A first pass at `menu()` or `getnights()` might use a simple `scanf()` without the data-verification features that have been added. Then, after the basic version works, you can begin improving each module.

Finding Addresses: The & Operator

One of the most important C concepts (and sometimes one of the most perplexing) is the *pointer*, which is a variable used to store an address. You've already seen that `scanf()` uses addresses for arguments. More generally, any C function that modifies a value in the calling function without using a `return` value uses addresses. We'll cover functions using addresses next, beginning with the unary `&` operator. (The next chapter continues the exploration and exploitation of pointers.)

The unary `&` operator gives you the address where a variable is stored. If `pooh` is the name of a variable, `&pooh` is the address of the variable. You can think of the address as a location in memory. Suppose you have the following statement:

```
pooh = 24;
```

Suppose that the address where `pooh` is stored is `0B76`. (PC addresses often are given as hexadecimal values.) Then the statement

```
printf("%d %p\n", pooh, &pooh);
```

would produce this (`%p` is the specifier for addresses):

```
24 0B76
```

Listing 9.12 uses this operator to see where variables of the same name—but in different functions—are kept.

Listing 9.12 **The `loccheck.c` Program**

```
/* loccheck.c  -- checks to see where variables are stored  */
#include <stdio.h>
void mikado(int);                      /* declare function  */
int main(void)
{
    int pooh = 2, bah = 5;             /* local to main()   */
```

```
    printf("In main(), pooh = %d and &pooh = %p\n",
            pooh, &pooh);
    printf("In main(), bah = %d and &bah = %p\n",
            bah, &bah);
    mikado(pooh);

    return 0;
}

void mikado(int bah)                    /* define function   */
{
    int pooh = 10;                      /* local to mikado() */

    printf("In mikado(), pooh = %d and &pooh = %p\n",
            pooh, &pooh);
    printf("In mikado(), bah = %d and &bah = %p\n",
            bah, &bah);
}
```

Listing 9.12 uses the ANSI C `%p` format for printing the addresses. Our system produced the following output for this little exercise:

```
In main(), pooh = 2 and &pooh = 0x7fff5fbff8e8
In main(), bah = 5 and &bah = 0x7fff5fbff8e4
In mikado(), pooh = 10 and &pooh = 0x7fff5fbff8b8
In mikado(), bah = 2 and &bah = 0x7fff5fbff8bc
```

The way that `%p` represents addresses varies among implementations. However, many implementations, such as one used for this example, display the address in hexadecimal form. Incidentally, given that each hexadecimal digit corresponds to four bits, these 12-digit address correspond to 48-bit addresses.

What does this output show? First, the two `pooh`s have different addresses. The same is true for the two `bah`s. So, as promised, the computer considers them to be four separate variables. Second, the call `mikado(pooh)` did convey the value (`2`) of the actual argument (`pooh` of `main()`) to the formal argument (`bah` of `mikado()`). Note that just the value was transferred. The two variables involved (`pooh` of `main()` and `bah` of `mikado()`) retain their distinct identities.

We raise the second point because it is not true for all languages. In FORTRAN, for example, the subroutine affects the original variable in the calling routine. The subroutine's variable might have a different name, but the address is the same. C doesn't do this. Each function uses its own variables. This is preferable because it prevents the original variable from being altered mysteriously by some side effect of the called function. However, it can make for some difficulties, too, as the next section shows.

Altering Variables in the Calling Function

Sometimes you want one function to make changes in the variables of a different function. For example, a common task in sorting problems is interchanging the values of two variables. Suppose you have two variables called `x` and `y` and you want to swap their values. The simple sequence

```
x = y;
y = x;
```

does not work because by the time the second line is reached, the original value of `x` has already been replaced by the original `y` value. An additional line is needed to temporarily store the original value of `x`.

```
temp = x;
x = y;
y = temp;
```

Now that the method works, you can put it into a function and construct a driver to test it. To make clear which variables belong to `main()` and which belong to the `interchange()` function, Listing 9.13 uses `x` and `y` for the first, and `u` and `v` for the second.

Listing 9.13 **The `swap1.c` Program**

```
/* swap1.c -- first attempt at a swapping function */
#include <stdio.h>
void interchange(int u, int v); /* declare function */

int main(void)
{
    int x = 5, y = 10;

    printf("Originally x = %d and y = %d.\n", x , y);
    interchange(x, y);
    printf("Now x = %d and y = %d.\n", x, y);

    return 0;
}

void interchange(int u, int v)  /* define function  */
{
    int temp;

    temp = u;
    u = v;
    v = temp;
}
```

Running the program gives these results:

```
Originally x = 5 and y = 10.
Now x = 5 and y = 10.
```

Oops! The values didn't get switched! Let's put some print statements into `interchange()` to see what has gone wrong (see Listing 9.14).

Listing 9.14 **The `swap2.c` Program**

```
/* swap2.c -- researching swap1.c */
#include <stdio.h>
void interchange(int u, int v);

int main(void)
{
    int x = 5, y = 10;

    printf("Originally x = %d and y = %d.\n", x , y);
    interchange(x, y);
    printf("Now x = %d and y = %d.\n", x, y);

    return 0;
}

void interchange(int u, int v)
{
    int temp;

    printf("Originally u = %d and v = %d.\n", u , v);
    temp = u;
    u = v;
    v = temp;
    printf("Now u = %d and v = %d.\n", u, v);
}
```

Here is the new output:

```
Originally x = 5 and y = 10.
Originally u = 5 and v = 10.
Now u = 10 and v = 5.
Now x = 5 and y = 10.
```

Nothing is wrong with `interchange()`; it does swap the values of `u` and `v`. The problem is in communicating the results to `main()`. As we pointed out, `interchange()` uses different variables from `main()`, so interchanging the values of `u` and `v` has no effect on `x` and `y`! Can you somehow use `return`? Well, you could finish `interchange()` with the line

```
return(u);
```

and then change the call in `main()` to this:

```
x = interchange(x,y);
```

This change gives `x` its new value, but it leaves `y` in the cold. With `return`, you can send just one value back to the calling function, but you need to communicate two values. It can be done! All you have to do is use pointers.

Pointers: A First Look

Pointers? What are they? Basically, a *pointer* is a variable (or, more generally, a data object) whose value is a memory address. Just as a `char` variable has a character as a value and an `int` variable has an integer as a value, the pointer variable has an address as a value. Pointers have many uses in C; in this chapter, you'll see how and why they are used as function parameters.

If you give a particular pointer variable the name `ptr`, you can have statements such as the following:

```
ptr = &pooh;  // assigns pooh's address to ptr
```

We say that `ptr` "points to" `pooh`. The difference between `ptr` and `&pooh` is that `ptr` is a variable, and `&pooh` is a constant. Or, `ptr` is a modifiable lvalue and `&pooh` is an rvalue. If you want, you can make `ptr` point elsewhere:

```
ptr = &bah;  // make ptr point to bah instead of to pooh
```

Now the value of `ptr` is the address of `bah`.

To create a pointer variable, you need to be able to declare its type. Suppose you want to declare `ptr` so that it can hold the address of an `int`. To make this declaration, you need to use a new operator. Let's examine that operator now.

The Indirection Operator: *

Suppose you know that `ptr` points to `bah`, as shown here:

```
ptr = &bah;
```

Then you can use the *indirection* operator `*` (also called the *dereferencing* operator) to find the value stored in `bah` (don't confuse this unary indirection operator with the binary `*` operator of multiplication—same symbol, different syntax):

```
val = *ptr;  // finding the value ptr points to
```

The statements `ptr = &bah;` and `val = *ptr;` taken together amount to the following statement:

```
val = bah;
```

Using the address and indirection operators is a rather indirect way of accomplishing this result, hence the name "indirection operator."

Summary: Pointer-Related Operators

The Address Operator:

`&`

General Comments:

When followed by a variable name, `&` gives the address of that variable.

Example:

`&nurse` is the address of the variable `nurse`.

The Indirection Operator: `*`

General Comments:

When followed by a pointer name or an address, `*` gives the value stored at the pointed-to address.

Example:

```
nurse = 22;
ptr = &nurse;  // pointer to nurse
val = *ptr;    // assigns value at location ptr to val
```

The net effect is to assign the value `22` to `val`.

Declaring Pointers

You already know how to declare `int` variables and other fundamental types. How do you declare a pointer variable? You might guess that the form is like this:

```
pointer ptr;     // not the way to declare a pointer/
```

Why not? Because it is not enough to say that a variable is a pointer. You also have to specify the kind of variable to which the pointer points. The reason is that different variable types take up different amounts of storage, and some pointer operations require knowledge of that storage size. Also, the program has to know what kind of data is stored at the address. A `long` and a `float` might use the same amount of storage, but they store numbers quite differently. Here's how pointers are declared:

```
int * pi;         // pi is a pointer to an integer variable
char * pc;        // pc is a pointer to a character variable
float * pf, * pg; // pf, pg are pointers to float variables
```

The type specification identifies the type of variable pointed to, and the asterisk (`*`) identifies the variable itself as a pointer. The declaration `int * pi;` says that `pi` is a pointer and that `*pi` is type `int` (see Figure 9.5).

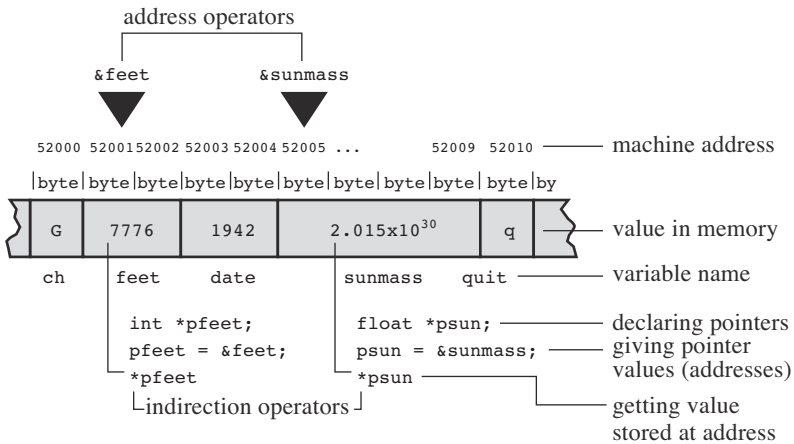

Figure 9.5 Declaring and using pointers.

The space between the * and the pointer name is optional. Often, programmers use the space in a declaration and omit it when dereferencing a variable.

The value (*pc) of what pc points to is of type char. What of pc itself? We describe it as being of type "pointer to char." The value of pc is an address, and it is represented internally as an unsigned integer on most systems. However, you shouldn't think of a pointer as an integer type. There are things you can do with integers that you can't do with pointers, and vice versa. For example, you can multiply one integer by another, but you can't multiply one pointer by another. So a pointer really is a new type, not an integer type. Therefore, as mentioned before, ANSI C provides the %p form specifically for pointers.

Using Pointers to Communicate Between Functions

We have touched only the surface of the rich and fascinating world of pointers, but our concern here is using pointers to solve our communication problem. Listing 9.15 shows a program that uses pointers to make the interchange() function work. Let's look at it, run it, and then try to understand how it works.

Listing 9.15 **The swap3.c Program**

```
/* swap3.c -- using pointers to make swapping work */
#include <stdio.h>
void interchange(int * u, int * v);

int main(void)
{
    int x = 5, y = 10;
```

```
    printf("Originally x = %d and y = %d.\n", x, y);
    interchange(&x, &y);  // send addresses to function
    printf("Now x = %d and y = %d.\n", x, y);

    return 0;
}

void interchange(int * u, int * v)
{
    int temp;

    temp = *u;       // temp gets value that u points to
    *u = *v;
    *v = temp;
}
```

After all this build-up, does Listing 9.15 really work?

```
Originally x = 5 and y = 10.
Now x = 10 and y = 5.
```

Yes, it works.

Now, let's see how Listing 9.15 works. First, the function call looks like this:

```
interchange(&x, &y);
```

Instead of transmitting the *values* of `x` and `y`, the function transmits their *addresses*. That means the formal arguments `u` and `v`, appearing in the prototype and in the definition of `interchange()`, will have addresses as their values. Therefore, they should be declared as pointers. Because `x` and `y` are integers, `u` and `v` are pointers to integers, so declare them as follows:

```
void interchange (int * u, int * v)
```

Next, the body of the function declares

```
int temp;
```

to provide the needed temporary storage. To store the value of `x` in `temp`, use

```
temp = *u;
```

Remember, `u` has the value `&x`, so `u` points to `x`. This means that `*u` gives you the value of `x`, which is what we want. Don't write

```
temp = u;   /* NO */
```

because that would assign `temp` the address of `x` rather than its value, and we are trying to interchange values, not addresses.

Similarly, to assign the value of `y` to `x`, use

```
*u = *v;
```

which ultimately has this effect:

```
x = y;
```

Let's summarize what this example does. We want a function that alters the values `x` and `y`. By passing the function the addresses of `x` and `y`, we give `interchange()` access to those variables. Using pointers and the `*` operator, the function can examine the values stored at those locations and change them.

You can omit the variable names in the ANSI C prototype. Then the prototype declaration looks like this:

```
void interchange(int *, int *);
```

In general, you can communicate two kinds of information about a variable to a function. If you use a call of the form

```
function1(x);
```

you transmit the value of `x`. If you use a call of the form

```
function2(&x);
```

you transmit the address of `x`. The first form requires that the function definition includes a formal argument of the same type as `x`:

```
int function1(int num)
```

The second form requires the function definition to include a formal parameter that is a pointer to the right type:

```
int function2(int * ptr)
```

Use the first form if the function needs a value for some calculation or action. Use the second form if the function needs to alter variables in the calling function. You have been doing this all along with the `scanf()` function. When you want to read in a value for a variable (`num`, for example), you use `scanf("%d", &num)`. That function reads a value and then uses the address you give it to store the value.

Pointers enable you to get around the fact that the variables of `interchange()` are local. They let that function reach out into `main()` and alter what is stored there.

Pascal and Modula-2 users might recognize the first form as being the same as Pascal's value parameter and the second form as being similar (but not identical) to Pascal's variable parameter. C++ users will recognize pointer variables and wonder if C, like C++, also has reference variables. The answer to that question is no. BASIC users might find the whole setup a bit unsettling. If this section seems strange to you, be assured that a little practice will make at least some uses of pointers seem simple, normal, and convenient (see Figure 9.6).

Variables: Names, Addresses, and Values

The preceding discussion of pointers has hinged on the relationships between the names, addresses, and values of variables. Let's discuss these matters further.

When you write a program, you can think of a variable as having two attributes: a name and a value. (There are other attributes, including type, but that's another matter.) After the program has been compiled and loaded, the computer also thinks of the same variable as having two attributes: an address and a value. An address is the computer's version of a name.

In many languages, the address is the computer's business, concealed from the programmer. In C, however, you can access the address through the `&` operator.

For example, `&barn` is the address of the variable `barn`.

You can get the value from the name just by using the name.

For example, `printf("%d\n", barn)` prints the value of `barn`.

You can get the value from the address by using the `*` operator.

Given `pbarn = &barn;`, `*pbarn` is the value stored at address `&barn`.

In short, a regular variable makes the value the primary quantity and the address a derived quantity, via the `&` operator. A pointer variable makes the address the primary quantity and the value a derived quantity via the `*` operator.

Although you can print an address to satisfy your curiosity, that is not the main use for the `&` operator. More important, using `&`, `*`, and pointers enables you to manipulate addresses and their contents symbolically, as in `swap3.c` (Listing 9.15).

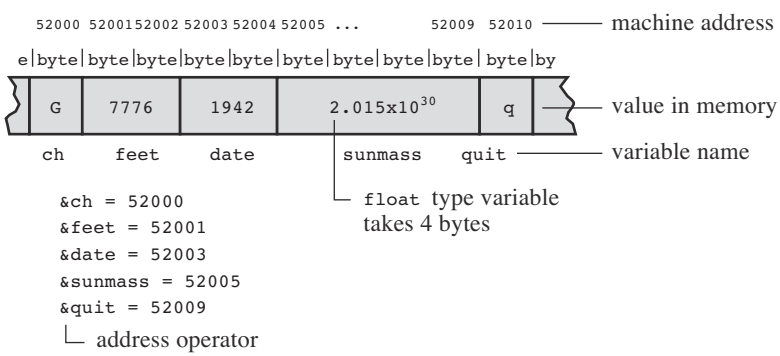

Figure 9.6 Names, addresses, and values in a byte-addressable system, such as a PC.

Summary: Functions

Form:

A typical ANSI C function definition has this form:

```
return-type name(parameter declaration list)
function body
```

The argument declaration list is a comma-separated list of variable declarations. Variables other than the function parameters are declared within the body, which is bounded by braces.

Example:

```
int diff(int x, int y)      // ANSI C
{                           // begin function body
    int z;                  // declare local variable

    z = x - y;

    return z;               // return a value
}                           // end function body
```

Communicating Values:

Arguments are used to convey values from the calling function to the function. If variables `a` and `b` have the values `5` and `2`, the call

```
c = diff(a,b);
```

transmits `5` and `2` to the variables `x` and `y`. The values `5` and `2` are called *actual arguments*, and the `diff()` variables `x` and `y` are called *formal parameters*. The keyword `return` communicates one value from the function to the calling function. In this example, `c` receives the value of `z`, which is `3`. A function ordinarily has no effect on the variables in a calling function. To directly affect variables in the calling function, use pointers as arguments. This might be necessary if you want to communicate more than one value back to the calling function.

Function Return Type:

The function return type indicates the type of value the function returns. If the returned value is of a type different from the declared return type, the value is type cast to the declared type.

Function Signature:

The function return type together with the function parameter list constitute the function signature. Thus, it specifies the types for values that go into the function and for the value that comes out of the function.

Example:

```
double duff(double, int);  // function prototype
int main(void)
{
    double q, x;
    int n;
```

```
...
     q = duff(x,n);     // function call


...
 }

double duff(double u, int k)  // function definition
{
    double tor;
...
    return tor;  // returns a double value
}
```

Key Concepts

If you want to program successfully and efficiently in C, you need to understand functions. It's useful, even essential, to organize larger programs into several functions. If you follow the practice of giving one function one task, your programs will be easier to understand and debug. Make sure that you understand how functions communicate information to one another—that is, that you understand how function arguments and return values work. Also, be aware how function parameters and other local variables are private to a function; thus, declaring two variables of the same name in different functions creates two distinct variables. Also, one function does not have direct access to variables declared in another function. This limited access helps preserve data integrity. However, if you do need one function to access another function's data, you can use pointer function arguments.

Summary

Use functions as building blocks for larger programs. Each function should have a single, well-defined purpose. Use arguments to communicate values to a function, and use the keyword `return` to communicate back a value. If the function returns a value not of type `int`, you must specify the function type in the function definition and in the declaration section of the calling function. If you want the function to affect variables in the calling function, use addresses and pointers.

ANSI C offers *function prototyping*, a powerful C enhancement that allows compilers to verify that the proper number and types of arguments are used in a function call.

A C function can call itself; this is called *recursion*. Some programming problems lend themselves to recursive solutions, but recursion can be inefficient in its use of memory and time.

Review Questions

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

1. What is the difference between an actual argument and a formal parameter?

2. Write ANSI C function headings for the following functions described. Note we are asking just for the headings, not the body.
   a. `donut()` takes an `int` argument and prints that number of `0`s.
   b. `gear()` takes two `int` arguments and returns type `int`.
   c. `guess()` takes no arguments and returns an `int` value.
   d. `stuff_it()` takes a `double` and the address of a `double` variable and stores the first value in the given location.

3. Write ANSI C function headings for the following functions described. Note that you need write only the headings, not the body.
   a. `n_to_char()` takes an `int` argument and returns a `char`.
   b. `digits()` takes a `double` argument and an `int` argument and returns an `int`.
   c. `which()` takes two addresses of double as arguments and returns the address of a `double`.
   d. `random()` takes no argument and returns an `int`.

4. Devise a function that returns the sum of two integers.

5. What changes, if any, would you need to make to have the function of question 4 add two `double` numbers instead?

6. Devise a function called `alter()` that takes two `int` variables, `x` and `y`, and changes their values to their sum and their difference, respectively.

7. Is anything wrong with this function definition?

```
void salami(num)
{
   int num, count;

   for (count = 1; count <= num; num++)
       printf(" O salami mio!\n");
}
```

8. Write a function that returns the largest of three integer arguments.

9. Given the following output:

```
Please choose one of the following:
1) copy files            2) move files
3) remove files          4) quit
Enter the number of your choice:
```

   a. Write a function that displays a menu of four numbered choices and asks you to choose one. (The output should look like the preceding.)

   b. Write a function that has two `int` arguments: a lower limit and an upper limit. The function should read an integer from input. If the integer is outside the limits, the function should print a menu again (using the function from part "a" of this question) to reprompt the user and then get a new value. When an integer in the proper limits is entered, the function should return that value to the calling function. Entering a noninteger should cause the function to return the quit value of `4`.

   c. Write a minimal program using the functions from parts "a" and "b" of this question. By *minimal,* we mean it need not actually perform the actions promised by the menu; it should just show the choices and get a valid response.

Programming Exercises

1. Devise a function called `min(x,y)` that returns the smaller of two `double` values. Test the function with a simple driver.

2. Devise a function `chline(ch,i,j)` that prints the requested character in columns `i` through `j`. Test it in a simple driver.

3. Write a function that takes three arguments: a character and two integers. The character is to be printed. The first integer specifies the number of times that the character is to be printed on a line, and the second integer specifies the number of lines that are to be printed. Write a program that makes use of this function.

4. The harmonic mean of two numbers is obtained by taking the inverses of the two numbers, averaging them, and taking the inverse of the result. Write a function that takes two `double` arguments and returns the harmonic mean of the two numbers.

5. Write and test a function called `larger_of()` that replaces the contents of two `double` variables with the maximum of the two values. For example, `larger_of(x,y)` would reset both `x` and `y` to the larger of the two.

6. Write and test a function that takes the addresses of three `double` variables as arguments and that moves the value of the smallest variable into the first variable, the middle value to the second variable, and the largest value into the third variable.

7. Write a program that reads characters from the standard input to end-of-file. For each character, have the program report whether it is a letter. If it is a letter, also report its numerical location in the alphabet. For example, *c* and *C* would both be letter 3. Incorporate a function that takes a character as an argument and returns the numerical location if the character is a letter and that returns –1 otherwise.

8. Chapter 6, "C Control Statements: Looping," (Listing 6.20) shows a `power()` function that returned the result of raising a type `double` number to a positive integer value. Improve the function so that it correctly handles negative powers. Also, build into the function that 0 to any power other than 0 is 0 and that any number to the 0 power is 1. (It should report that 0 to the 0 is undefined, then say it's using a value of 1.) Use a loop. Test the function in a program.

9. Redo Programming Exercise 8, but this time use a recursive function.

10. Generalize the `to_binary()` function of Listing 9.8 to a `to_base_n()` function that takes a second argument in the range 2–10. It then should print the number that is its first argument to the number base given by the second argument. For example, `to_base_n(129,8)` would display `201`, the base-8 equivalent of `129`. Test the function in a complete program.

11. Write and test a `Fibonacci()` function that uses a loop instead of recursion to calculate Fibonacci numbers.