Functions

CHAPTER

5

Assist. Prof. Dr.
Ahmed Hashim Mohammed

IN THIS CHAPTER

A function groups a number of program statements into a unit and gives it a name. This unit can then be invoked from other parts of the program.

The most important reason to use functions is to aid in the conceptual organization of a program. Dividing a program into functions is, as we discussed in Chapter 1, “The Big Picture,” one of the major principles of structured programming. (However, object-oriented programming provides additional, more powerful ways to organize programs.)

Another reason to use functions (and the reason they were invented, long ago) is to reduce program size. Any sequence of instructions that appears in a program more than once is a candidate for being made into a function. The function’s code is stored in only one place in memory, even though the function is executed many times in the course of the program. Figure 5.1 shows how a function is invoked from different sections of a program.

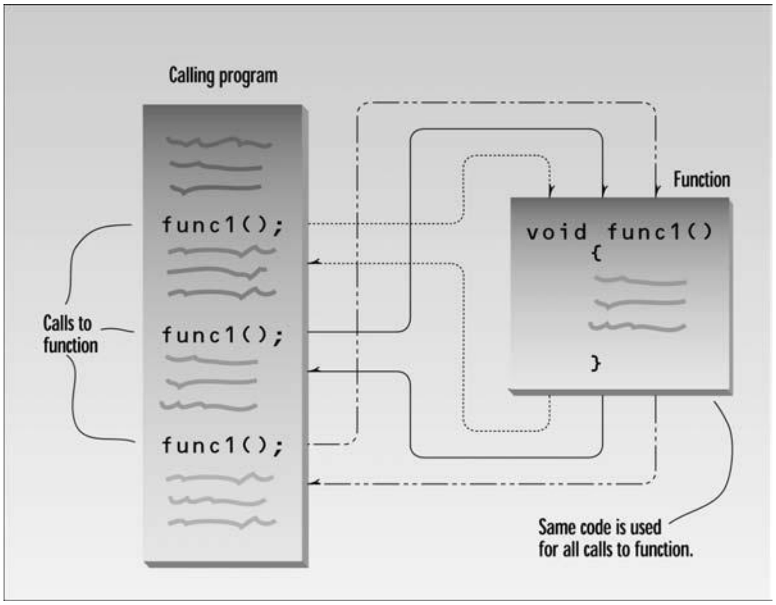


FIGURE 5.1
Flow of control to a function.

Functions in C++ (and C) are similar to subroutines and procedures in various other languages.

Simple Functions

Our first example demonstrates a simple function whose purpose is to print a line of 45 asterisks. The example program generates a table, and lines of asterisks are used to make the table more readable. Here’s the listing for TABLE:

```
// table.cpp
// demonstrates simple function
#include <iostream>
using namespace std;

void starline();                          //function declaration
                                          //   (prototype)
int main()
   {
   starline();                            //call to function
   cout << “Data type   Range” << endl;
   starline();                            //call to function
   cout << “char        -128 to 127” << endl
        << “short       -32,768 to 32,767” << endl
        << “int         System dependent” << endl
        << “long        -2,147,483,648 to 2,147,483,647” << endl;
   starline();                            //call to function
   return 0;
   }
//--------------------------------------------------------------
// starline()
// function definition
void starline()                           //function declarator
   {
   for(int j=0; j<45; j++)                //function body
      cout << ‘*’;
   cout << endl;
   }
```

The output from the program looks like this:

```
*********************************************
Data type   Range
*********************************************
char        -128 to 127
short       -32,768 to 32,767
int         System dependent
long     -2,147,483,648 to 2,147,483,647
*********************************************
```

The program consists of two functions: `main()` and `starline()`. You’ve already seen many programs that use `main()` alone. What other components are necessary to add a function to the program? There are three: the function *declaration*, the *calls* to the function, and the function *definition*.

The Function Declaration

Just as you can’t use a variable without first telling the compiler what it is, you also can’t use a function without telling the compiler about it. There are two ways to do this. The approach we show here is to *declare* the function before it is called. (The other approach is to *define* it before it’s called; we’ll examine that next.) In the TABLE program, the function `starline()` is declared in the line

```
void starline();
```

The declaration tells the compiler that at some later point we plan to present a function called *starline*. The keyword `void` specifies that the function has no return value, and the empty parentheses indicate that it takes no arguments. (You can also use the keyword `void` in parentheses to indicate that the function takes no arguments, as is often done in C, but leaving them empty is the more common practice in C++.) We’ll have more to say about arguments and return values soon.

Notice that the function declaration is terminated with a semicolon. It is a complete statement in itself.

Function declarations are also called *prototypes*, since they provide a model or blueprint for the function. They tell the compiler, “a function that looks like this is coming up later in the program, so it’s all right if you see references to it before you see the function itself.” The information in the declaration (the return type and the number and types of any arguments) is also sometimes referred to as the function *signature*.

Calling the Function

The function is *called* (or *invoked*, or *executed*) three times from `main()`. Each of the three calls looks like this:

```
starline();
```

This is all we need to call the function: the function name, followed by parentheses. The syntax of the call is very similar to that of the declaration, except that the return type is not used. The call is terminated by a semicolon. Executing the call statement causes the function to execute; that is, control is transferred to the function, the statements in the function definition (which we’ll examine in a moment) are executed, and then control returns to the statement following the function call.

The Function Definition

Finally we come to the function itself, which is referred to as the function *definition*. The definition contains the actual code for the function. Here’s the definition for `starline()`:

```
void starline()                  //declarator
   {
   for(int j=0; j<45; j++)     //function body
      cout << '*';
   cout << endl;
   }
```

The definition consists of a line called the *declarator*, followed by the function *body*. The function body is composed of the statements that make up the function, delimited by braces.

The declarator must agree with the declaration: It must use the same function name, have the same argument types in the same order (if there are arguments), and have the same return type.

Notice that the declarator is *not* terminated by a semicolon. Figure 5.2 shows the syntax of the function declaration, function call, and function definition.

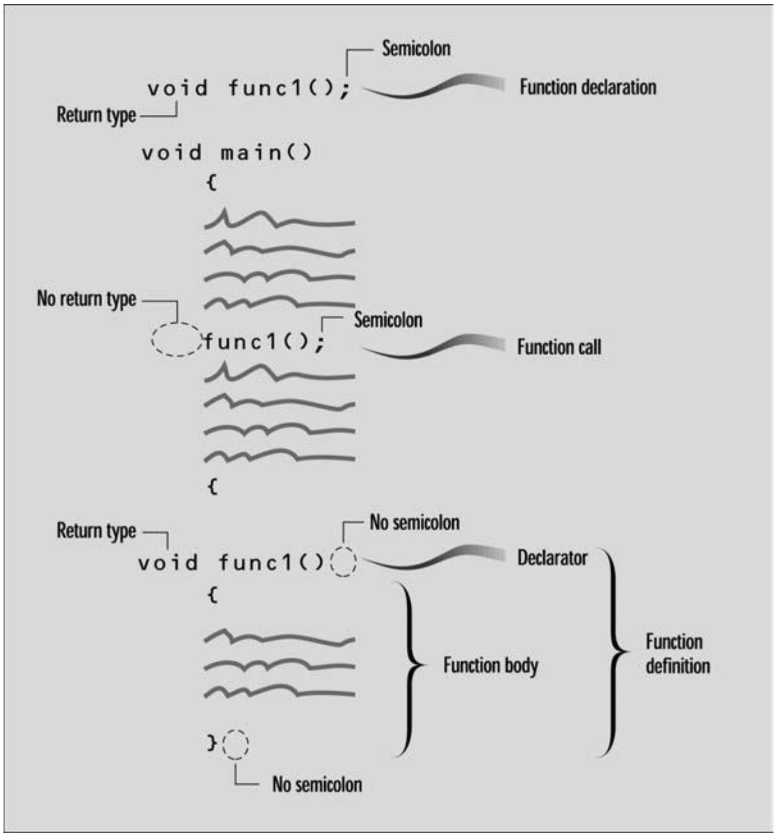


FIGURE 5.2
Function syntax.

When the function is called, control is transferred to the first statement in the function body. The other statements in the function body are then executed, and when the closing brace is encountered, control returns to the calling program.

Table 5.1 summarizes the different function components.

TABLE 5.1 Function Components

Component	*Purpose*	*Example*
Declaration (prototype)	Specifies function name, argument types, and return value. Alerts compiler (and programmer) that a function is coming up later.	`void func();`
Call	Causes the function to be executed.	`func();`
Definition	The function itself. Contains the lines of code that constitute the function.	`void func()` `{` `// lines of code` `}`
Declarator	First line of definition.	`void func()`

Comparison with Library Functions

We've already seen some library functions in use. We have embedded calls to library functions, such as

```
ch = getche();
```

in our program code. Where are the declaration and definition for this library function? The declaration is in the header file specified at the beginning of the program (CONIO.H, for `getche()`). The definition (compiled into executable code) is in a library file that's linked automatically to your program when you build it.

When we use a library function we don't need to write the declaration or definition. But when we write our own functions, the declaration and definition are part of our source file, as we've shown in the TABLE example. (Things get more complicated in multifile programs, as we'll discuss in Chapter 13, "Multifile Programs.")

Eliminating the Declaration

The second approach to inserting a function into a program is to eliminate the function declaration and place the function definition (the function itself) in the listing before the first call to the function. For example, we could rewrite TABLE to produce TABLE2, in which the definition for `starline()` appears first.

```
// table2.cpp
// demonstrates function definition preceding function calls
#include <iostream>
using namespace std;                 //no function declaration
//----------------------------------------------------------------
// starline()                        //function definition
void starline()
   {
   for(int j=0; j<45; j++)
      cout << ‘*’;
   cout << endl;
   }
//----------------------------------------------------------------
int main()                           //main() follows function
   {
   starline();                       //call to function
   cout << “Data type   Range” << endl;
   starline();                       //call to function
   cout << “char        -128 to 127” << endl
        << “short       -32,768 to 32,767” << endl
        << “int         System dependent” << endl
        << “long        -2,147,483,648 to 2,147,483,647” << endl;
   starline();                       //call to function
   return 0;
   }
```

This approach is simpler for short programs, in that it removes the declaration, but it is less flexible. To use this technique when there are more than a few functions, the programmer must give considerable thought to arranging the functions so that each one appears before it is called by any other. Sometimes this is impossible. Also, many programmers prefer to place `main()` first in the listing, since it is where execution begins. In general we'll stick with the first approach, using declarations and starting the listing with `main()`.

Passing Arguments to Functions

An *argument* is a piece of data (an `int` value, for example) passed from a program to the function. Arguments allow a function to operate with different values, or even to do different things, depending on the requirements of the program calling it.

Passing Constants

As an example, let's suppose we decide that the `starline()` function in the last example is too rigid. Instead of a function that always prints 45 asterisks, we want a function that will print any character any number of times.

Here’s a program, TABLEARG, that incorporates just such a function. We use arguments to pass the character to be printed and the number of times to print it.

```
// tablearg.cpp
// demonstrates function arguments
#include <iostream>
using namespace std;
void repchar(char, int);                  //function declaration

int main()
   {
   repchar(‘-’, 43);                      //call to function
   cout << “Data type   Range” << endl;
   repchar(‘=’, 23);                      //call to function
   cout << “char        -128 to 127” << endl
        << “short       -32,768 to 32,767” << endl
        << “int         System dependent” << endl
        << “double      -2,147,483,648 to 2,147,483,647” << endl;
   repchar(‘-’, 43);                      //call to function
   return 0;
   }
//--------------------------------------------------------------
// repchar()
// function definition
void repchar(char ch, int n)              //function declarator
   {
   for(int j=0; j<n; j++)                 //function body
      cout << ch;
   cout << endl;
   }
```

The new function is called `repchar()`. Its declaration looks like this:

```
void repchar(char, int);   // declaration specifies data types
```

The items in the parentheses are the data types of the arguments that will be sent to `repchar()`: `char` and `int`.

In a function call, specific values—constants in this case—are inserted in the appropriate place in the parentheses:

```
repchar(‘-’, 43);   // function call specifies actual values
```

This statement instructs `repchar()` to print a line of 43 dashes. The values supplied in the call must be of the types specified in the declaration: the first argument, the `-` character, must be of type `char`; and the second argument, the number 43, must be of type `int`. The types in the declaration and the definition must also agree.

The next call to `repchar()`

```
repchar('=', 23);
```

tells it to print a line of 23 equal signs. The third call again prints 43 dashes. Here's the output from TABLEARG:

```
-------------------------------------------
Data type   Range
=======================
char        -128 to 127
short       -32,768 to 32,767
int         System dependent
long         -2,147,483,648 to 2,147,483,647
-------------------------------------------
```

The calling program supplies *arguments*, such as `'-'` and `43`, to the function. The variables used within the function to hold the argument values are called *parameters*; in `repchar()` they are `ch` and `n`. (We should note that many programmers use the terms argument and parameter somewhat interchangeably.) The declarator in the function definition specifies both the data types and the names of the parameters:

```
void repchar(char ch, int n)  //declarator specifies parameter
                              //names and data types
```

These parameter names, `ch` and `n`, are used in the function as if they were normal variables. Placing them in the declarator is equivalent to defining them with statements like

```
char ch;
int n;
```

When the function is called, its parameters are automatically initialized to the values passed by the calling program.

Passing Variables

In the TABLEARG example the arguments were constants: `'-'`, `43`, and so on. Let's look at an example where variables, instead of constants, are passed as arguments. This program, VARARG, incorporates the same `repchar()` function as did TABLEARG, but lets the user specify the character and the number of times it should be repeated.

```
// vararg.cpp
// demonstrates variable arguments
#include <iostream>
using namespace std;
void repchar(char, int);                  //function declaration
```

```
int main()
   {
   char chin;
   int nin;

   cout << “Enter a character: “;
   cin >> chin;
   cout << “Enter number of times to repeat it: “;
   cin >> nin;
   repchar(chin, nin);
   return 0;
   }
//--------------------------------------------------------------
// repchar()
// function definition
void repchar(char ch, int n)              //function declarator
   {
   for(int j=0; j<n; j++)                 //function body
      cout << ch;
   cout << endl;
   }
```

Here’s some sample interaction with VARARG:

```
Enter a character: +
Enter number of times to repeat it: 20
++++++++++++++++++++
```

Here `chin` and `nin` in `main()` are used as arguments to `repchar()`:

```
repchar(chin, nin);   // function call
```

The data types of variables used as arguments must match those specified in the function declaration and definition, just as they must for constants. That is, `chin` must be a `char`, and `nin` must be an `int`.

Passing by Value

In VARARG the particular values possessed by `chin` and `nin` when the function call is executed will be passed to the function. As it did when constants were passed to it, the function creates new variables to hold the values of these variable arguments. The function gives these new variables the names and data types of the parameters specified in the declarator: `ch` of type `char` and `n` of type `int`. It initializes these parameters to the values passed. They are then accessed like other variables by statements in the function body.

Passing arguments in this way, where the function creates copies of the arguments passed to it, is called *passing by value*. We'll explore another approach, *passing by reference*, later in this chapter. Figure 5.3 shows how new variables are created in the function when arguments are passed by value.

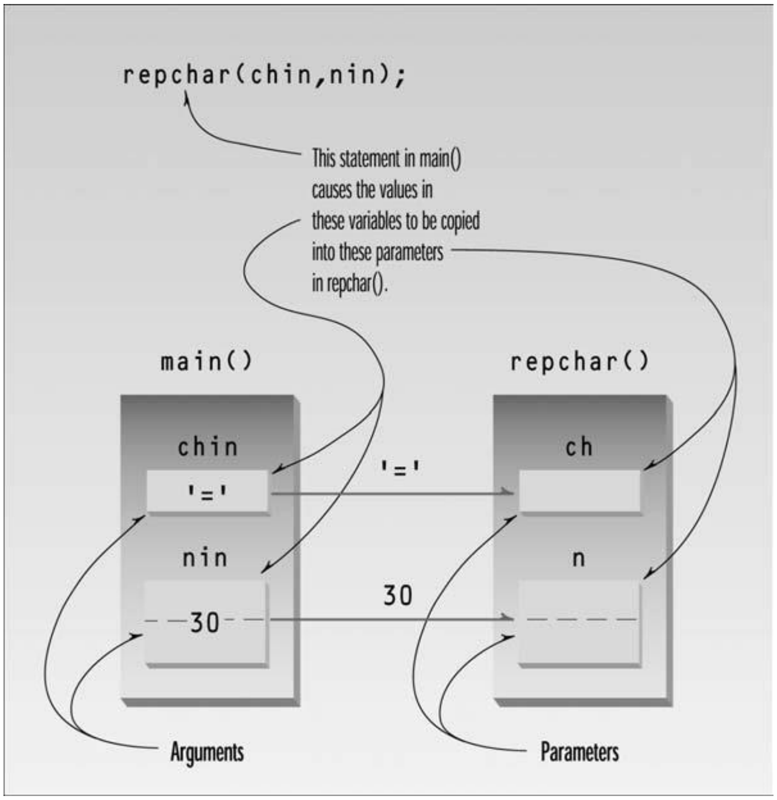

repchar(chin,nin);
This statement in main() causes the values in these variables to be copied into these parameters in repchar().
main()
repchar()
chin
'='
nin
30
ch
n
'='
30
Arguments
Parameters

FIGURE 5.3
Passing by value.

Structures as Arguments

Entire structures can be passed as arguments to functions. We'll show two examples, one with the `Distance` structure, and one with a structure representing a graphics shape.

Passing a Distance Structure

This example features a function that uses an argument of type `Distance`, the same structure type we saw in several programs in Chapter 4, "Structures." Here's the listing for ENGLDISP:

```
// engldisp.cpp
// demonstrates passing structure as argument
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
struct Distance               //English distance
   {
   int feet;
   float inches;
   };
////////////////////////////////////////////////////////////////
void engldisp( Distance );     //declaration

int main()
   {
   Distance d1, d2;            //define two lengths

                               //get length d1 from user
   cout << “Enter feet: “;  cin >> d1.feet;
   cout << “Enter inches: “;  cin >> d1.inches;

                               //get length d2 from user
   cout << “\nEnter feet: “;  cin >> d2.feet;
   cout << “Enter inches: “;  cin >> d2.inches;

   cout << “\nd1 = “;
   engldisp(d1);               //display length 1
   cout << “\nd2 = “;
   engldisp(d2);               //display length 2
   cout << endl;
   return 0;
   }
//--------------------------------------------------------------
// engldisp()
// display structure of type Distance in feet and inches
void engldisp( Distance dd )   //parameter dd of type Distance
   {
   cout << dd.feet << “\’-” << dd.inches << “\””;
   }
```

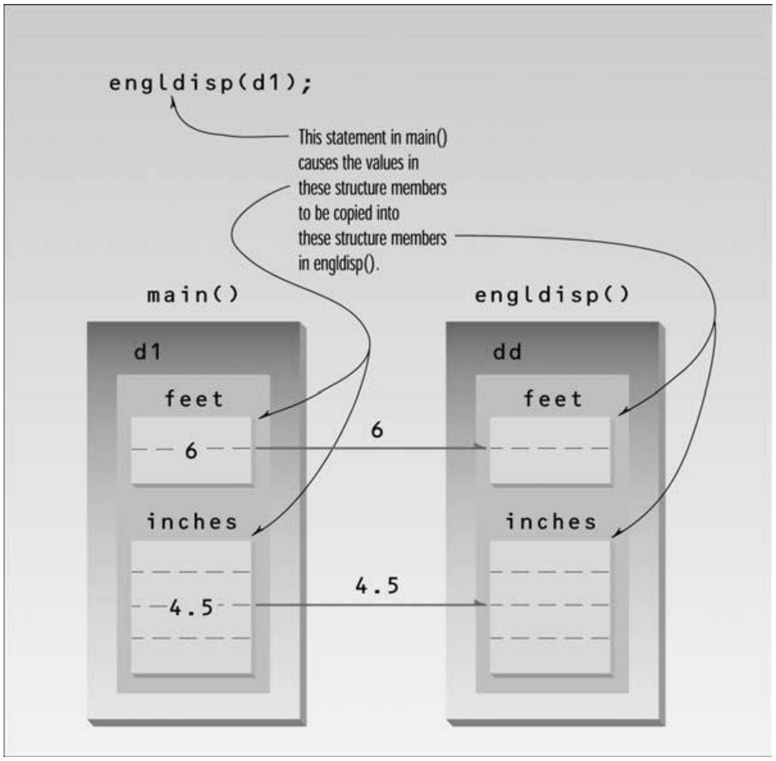
The `main()` part of this program accepts two distances in feet-and-inches format from the user, and places these values in two structures, `d1` and `d2`. It then calls a function, `engldisp()`, that takes a `Distance` structure variable as an argument. The purpose of the function is to display the distance passed to it in the standard format, such as 10'–2.25". Here’s some sample interaction with the program:

```
Enter feet: 6
Enter inches: 4
Enter feet: 5
Enter inches: 4.25

d1 = 6'-4"
d2 = 5'-4.25"
```

The function declaration and the function calls in `main()`, and the declarator in the function body, treat the structure variables just as they would any other variable used as an argument; this one just happens to be type `Distance`, rather than a basic type like `char` or `int`.

In `main()` there are two calls to the function `engldisp()`. The first passes the structure `d1`; the second passes `d2`. The function `engldisp()` uses a parameter that is a structure of type `Distance`, which it names `dd`. As with simple variables, this structure variable is automatically initialized to the value of the structure passed from `main()`. Statements in `engldisp()` can then access the members of `dd` in the usual way, with the expressions `dd.feet` and `dd.inches`. Figure 5.4 shows a structure being passed as an argument to a function.



engldisp(d1);
This statement in main() causes the values in these structure members to be copied into these structure members in engldisp().
main()
engldisp()
d1
dd
feet
feet
6
6
inches
inches
4.5
4.5

FIGURE 5.4
Structure passed as an argument.

As with simple variables, the structure parameter `dd` in `engldisp()` is not the same as the arguments passed to it (`d1` and `d2`). Thus, `engldisp()` could (although it doesn't do so here) modify `dd` without affecting `d1` and `d2`. That is, if `engldisp()` contained statements like

```
dd.feet = 2;
dd.inches = 3.25;
```

this would have no effect on `d1` or `d2` in `main()`.

Passing a `circle` Structure

The next example of passing a structure to a function makes use of the Console Graphics Lite functions. The source and header files for these functions are shown in Appendix E, "Console Graphics Lite," and can be downloaded from the publisher's Web site as described in the Introduction. You'll need to include the appropriate header file (MSOFTCON.H or BORLACON.H, depending on your compiler), and add the source file (MSOFTCON.CPP or BORLACON.CPP) to your project. The Console Graphics Lite functions are described in Appendix E, and the procedure for adding files to projects is described in Appendix C, "Microsoft Visual C++," and Appendix D, "Borland C++Builder."

In this example a structure called `circle` represents a circular shape. Circles are positioned at a certain place on the console screen, and have a certain radius. They also have a color and a fill pattern. Possible values for the colors and fill patterns can be found in Appendix E. Here's the listing for CIRCSTRC:

```
// circstrc.cpp
// circles as graphics objects
#include “msoftcon.h”         // for graphics functions
////////////////////////////////////////////////////////////////
struct circle                 //graphics circle
   {
   int xCo, yCo;              //coordinates of center
   int radius;
   color fillcolor;           //color
   fstyle fillstyle;          //fill pattern
   };
////////////////////////////////////////////////////////////////
void circ_draw(circle c)
   {
   set_color(c.fillcolor);               //set color
   set_fill_style(c.fillstyle);          //set fill pattern
   draw_circle(c.xCo, c.yCo, c.radius);  //draw solid circle
   }
//---------------------------------------------------------------
int main()
   {
   init_graphics();           //initialize graphics system
                              //create circles
   circle c1 = { 15, 7, 5, cBLUE, X_FILL };
   circle c2 = { 41, 12, 7, cRED, O_FILL };
   circle c3 = { 65, 18, 4, cGREEN, MEDIUM_FILL };

   circ_draw(c1);              //draw circles
   circ_draw(c2);
   circ_draw(c3);
   set_cursor_pos(1, 25);      //cursor to lower left corner
   return 0;
   }
```

The variables of type `circle`, which are `c1`, `c2`, and `c3`, are initialized to different sets of values. Here's how that looks for `c1`:

```
circle c1 = { 15, 7, 5, cBLUE, X_FILL };
```

We assume that your console screen has 80 columns and 25 rows. The first value in this definition, 15, is the *column number* (the x coordinate) and the 7 is the *row number* (the y coordinate, starting at the top of the screen) where the center of the circle will be located. The 5 is the radius of the circle, the `cBLUE` is its color, and the `X_FILL` constant means it will be filled with the letter X. The two other circles are initialized similarly.

Once all the circles are created and initialized, we draw them by calling the `circ_draw()` function three times, once for each circle. Figure 5.5 shows the output of the CIRCSTRC program. Admittedly the circles are a bit ragged; a result of the limited number of pixels in console-mode graphics.

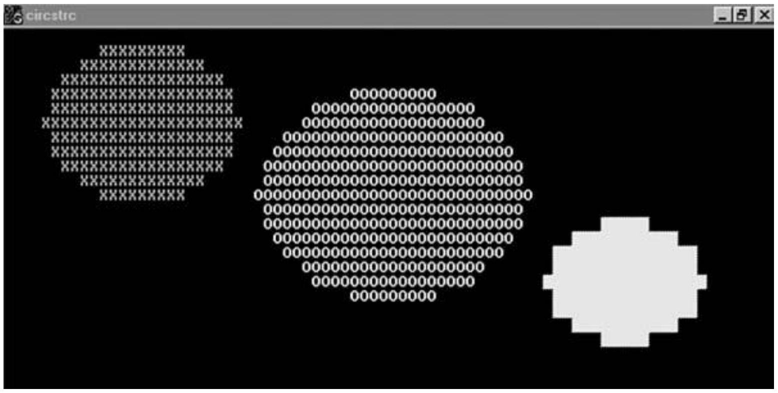


FIGURE 5.5
Output of the CIRCSTRC program.

Notice how the structure holds the characteristics of the circles, while the `circ_draw()` function causes them to actually do something (draw themselves). As we'll see in Chapter 6, "Objects and Classes," objects are formed by combining structures and functions to create entities that both possess characteristics and perform actions.

Names in the Declaration

Here's a way to increase the clarity of your function declarations. The idea is to insert meaningful names in the declaration, along with the data types. For example, suppose you were using a function that displayed a point on the screen. You could use a declaration with only data types

```
void display_point(int, int);  //declaration
```

but a better approach is

```
void display_point(int horiz, int vert);  //declaration
```

These two declarations mean exactly the same thing to the compiler. However, the first approach, with (`int`, `int`), doesn't contain any hint about which argument is for the vertical coordinate and which is for the horizontal coordinate. The advantage of the second approach is clarity for the programmer: Anyone seeing this declaration is more likely to use the correct arguments when calling the function.

Note that the names in the declaration have no effect on the names you use when calling the function. You are perfectly free to use any argument names you want:

```
display_point(x, y);  // function call
```

We'll use this name-plus-datatype approach when it seems to make the listing clearer.

Returning Values from Functions

When a function completes its execution, it can return a single value to the calling program. Usually this return value consists of an answer to the problem the function has solved. The next example demonstrates a function that returns a weight in kilograms after being given a weight in pounds. Here's the listing for CONVERT:

```
// convert.cpp
// demonstrates return values, converts pounds to kg
#include <iostream>
using namespace std;
float lbstokg(float);   //declaration

int main()
   {
   float lbs, kgs;
   cout << "\nEnter your weight in pounds: ";
   cin >> lbs;
   kgs = lbstokg(lbs);
   cout << "Your weight in kilograms is " << kgs << endl;
   return 0;
   }
//--------------------------------------------------------------
// lbstokg()
// converts pounds to kilograms
float lbstokg(float pounds)
   {
   float kilograms = 0.453592 * pounds;
   return kilograms;
   }
```

Here's some sample interaction with this program:

```
Enter your weight in pounds: 182
Your weight in kilograms is 82.553741
```

When a function returns a value, the data type of this value must be specified. The function declaration does this by placing the data type, `float` in this case, before the function name in the declaration and the definition. Functions in earlier program examples returned no value, so the return type was `void`. In the CONVERT program, the function `lbstokg()` (*pounds to kilograms*, where `lbs` means pounds) returns type `float`, so the declaration is

```
float lbstokg(float);
```

The first `float` specifies the return type. The `float` in parentheses specifies that an argument to be passed to `lbstokg()` is also of type `float`.

When a function returns a value, the call to the function

```
lbstokg(lbs)
```

is considered to be an expression that takes on the value returned by the function. We can treat this expression like any other variable; in this case we use it in an assignment statement:

```
kgs = lbstokg(lbs);
```

This causes the variable `kgs` to be assigned the value returned by `lbstokg()`.

The `return` Statement

The function `lbstokg()` is passed an argument representing a weight in pounds, which it stores in the parameter `pounds`. It calculates the corresponding weight in kilograms by multiplying this pounds value by a constant; the result is stored in the variable `kilograms`. The value of this variable is then returned to the calling program using a return statement:

```
return kilograms;
```

Notice that both `main()` and `lbstokg()` have a place to store the kilogram variable: `kgs` in `main()`, and `kilograms` in `lbstokg()`. When the function returns, the value in `kilograms` is *copied into* `kgs`. The calling program does not access the `kilograms` variable in the function; only the value is returned. This process is shown in Figure 5.6.

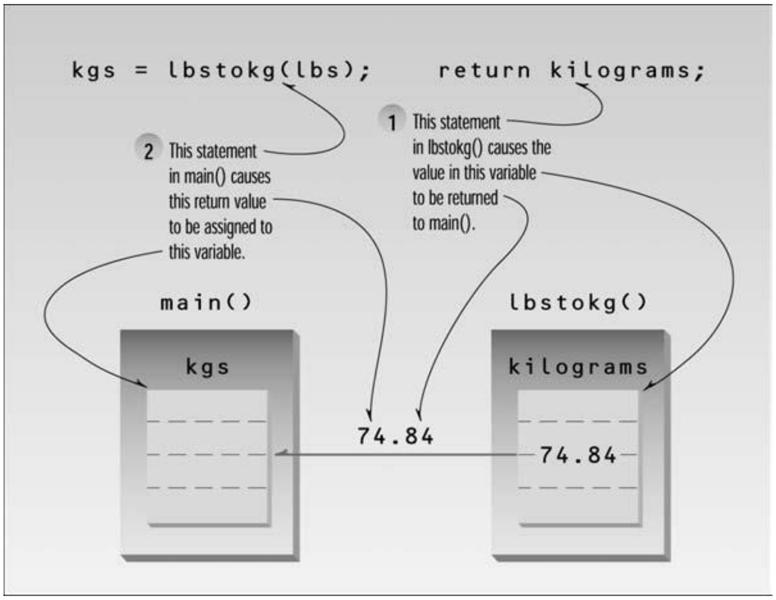

kgs = lbstokg(lbs);
return kilograms;
1 This statement in lbstokg() causes the value in this variable to be returned to main().
2 This statement in main() causes this return value to be assigned to this variable.
main()
lbstokg()
kgs
kilograms
74.84
74.84

FIGURE 5.6
Returning a value.

While many arguments may be sent to a function, only one argument may be returned from it. This is a limitation when you need to return more information. However, there are other approaches to returning multiple variables from functions. One is to pass arguments by reference, which we'll look at later in this chapter. Another is to return a structure with the multiple values as members, as we'll see soon.

You should always include a function's return type in the function declaration. If the function doesn't return anything, use the keyword `void` to indicate this fact. If you don't use a return type in the declaration, the compiler will assume that the function returns an `int` value. For example, the declaration

```
somefunc();   // declaration -- assumes return type is int
```

tells the compiler that `somefunc()` has a return type of `int`.

The reason for this is historical, based on usage in early versions of C. In practice, you shouldn't take advantage of this default type. Always specify the return type explicitly, even if it actually is `int`. This keeps the listing consistent and readable.

Eliminating Unnecessary Variables

The CONVERT program contains several variables that are used in the interest of clarity but are not really necessary. A variation of this program, CONVERT2, shows how expressions containing functions can often be used in place of variables.

```
// convert2.cpp
// eliminates unnecessary variables
#include <iostream>
using namespace std;
float lbstokg(float);   //declaration

int main()
   {
   float lbs;

   cout << "\nEnter your weight in pounds: ";
   cin >> lbs;
   cout << "Your weight in kilograms is " << lbstokg(lbs)
        << endl;
   return 0;
   }
//--------------------------------------------------------------
// lbstokg()
// converts pounds to kilograms
float lbstokg(float pounds)
   {
   return 0.453592 * pounds;
   }
```

In `main()` the variable `kgs` from the CONVERT program has been eliminated. Instead the function `lbstokg(lbs)` is inserted directly into the `cout` statement:

```
cout << "Your weight in kilograms is " << lbstokg(lbs) << endl;
```

Also in the `lbstokg()` function, the variable `kilograms` is no longer used. The expression `0.453592*pounds` is inserted directly into the `return` statement:

```
return 0.453592 * pounds;
```

The calculation is carried out and the resulting value is returned to the calling program, just as the value of a variable would be.

For clarity, programmers often put parentheses around the expression used in a return statement:

```
return (0.453592 * pounds);
```

Even when not required by the compiler, extra parentheses in an expression don’t do any harm, and they may help make the listing easier for us poor humans to read.

Experienced C++ (and C) programmers will probably prefer the concise form of CONVERT2 to the more verbose CONVERT. However, CONVERT2 is not so easy to understand, especially for the non-expert. The brevity-versus-clarity issue is a question of style, depending on your personal preference and on the expectations of those who will be reading your code.

Returning Structure Variables

We’ve seen that structures can be used as arguments to functions. You can also use them as return values. Here’s a program, RETSTRC, that incorporates a function that adds variables of type `Distance` and returns a value of this same type:

```
// retstrc.cpp
// demonstrates returning a structure
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
struct Distance                        //English distance
   {
   int feet;
   float inches;
   };
////////////////////////////////////////////////////////////////
Distance addengl(Distance, Distance);  //declarations
void engldisp(Distance);

int main()
   {
   Distance d1, d2, d3;                //define three lengths
                                       //get length d1 from user
   cout << “\nEnter feet: “;  cin >> d1.feet;
   cout << “Enter inches: “;  cin >> d1.inches;
                                       //get length d2 from user
   cout << “\nEnter feet: “;  cin >> d2.feet;
   cout << “Enter inches: “;  cin >> d2.inches;

   d3 = addengl(d1, d2);               //d3 is sum of d1 and d2
   cout << endl;
   engldisp(d1); cout << “ + “;        //display all lengths
   engldisp(d2); cout << " = ";
   engldisp(d3); cout << endl;
   return 0;
   }
//--------------------------------------------------------------
// addengl()
// adds two structures of type Distance, returns sum
Distance addengl( Distance dd1, Distance dd2 )
   {
   Distance dd3;                //define a new structure for sum

   dd3.inches = dd1.inches + dd2.inches; //add the inches
   dd3.feet = 0;                       //(for possible carry)
   if(dd3.inches >= 12.0)              //if inches >= 12.0,
      {                                //then decrease inches
      dd3.inches -= 12.0;              //by 12.0 and
      dd3.feet++;                      //increase feet
      }                                //by 1
   dd3.feet += dd1.feet + dd2.feet;    //add the feet
   return dd3;                         //return structure
   }
//--------------------------------------------------------------
// engldisp()
// display structure of type Distance in feet and inches
void engldisp( Distance dd )
   {
   cout << dd.feet << "\'-" << dd.inches << "\"";
   }
```

The program asks the user for two lengths, in feet-and-inches format, adds them together by calling the function `addengl()`, and displays the results using the `engldisp()` function introduced in the ENGLDISP program. Here's some output from the program:

```
Enter feet: 4
Enter inches: 5.5

Enter feet: 5
Enter inches: 6.5

4'-5.5" + 5'-6.5" = 10'-0"
```

The `main()` part of the program adds the two lengths, each represented by a structure of type `Distance`, by calling the function `addengl()`:

```
d3 = addengl(d1, d2);
```

This function returns the sum of `d1` and `d2`, in the form of a structure of type `Distance`. In `main()` the result is assigned to the structure `d3`.

Besides showing how structures are used as return values, this program also shows two functions (three if you count `main()`) used in the same program. You can arrange the functions in any order. The only rule is that the function declarations must appear in the listing before any calls are made to the functions.

Reference Arguments

A *reference* provides an *alias*—a different name—for a variable. One of the most important uses for references is in passing arguments to functions.

We've seen examples of function arguments passed by value. When arguments are passed by value, the called function creates a new variable of the same type as the argument and copies the argument's value into it. As we noted, the function cannot access the original variable in the calling program, only the copy it created. Passing arguments by value is useful when the function does not need to modify the original variable in the calling program. In fact, it offers insurance that the function cannot harm the original variable.

Passing arguments by reference uses a different mechanism. Instead of a value being passed to the function, a *reference to* the original variable, in the calling program, is passed. (It's actually the *memory address* of the variable that is passed, although you don't need to know this.)

An important advantage of passing by reference is that the function can access the actual variables in the calling program. Among other benefits, this provides a mechanism for passing more than one value from the function back to the calling program.

Passing Simple Data Types by Reference

The next example, REF, shows a simple variable passed by reference.

```
// ref.cpp
// demonstrates passing by reference
#include <iostream>
using namespace std;

int main()
   {
   void intfrac(float, float&, float&);      //declaration
   float number, intpart, fracpart;          //float variables

   do {
      cout << "\nEnter a real number: ";     //number from user
      cin >> number;
      intfrac(number, intpart, fracpart);    //find int and frac
      cout << "Integer part is " << intpart  //print them
           << ", fraction part is " << fracpart << endl;
      } while( number != 0.0 );              //exit loop on 0.0
   return 0;
   }
//--------------------------------------------------------------
// intfrac()
// finds integer and fractional parts of real number
void intfrac(float n, float& intp, float& fracp)
   {
   long temp = static_cast<long>(n); //convert to long,
   intp = static_cast<float>(temp);   //back to float
   fracp = n - intp;                  //subtract integer part
   }
```

The `main()` part of this program asks the user to enter a number of type `float`. The program will separate this number into an integer and a fractional part. That is, if the user's number is 12.456, the program should report that the integer part is 12.0 and the fractional part is 0.456. To find these two values, `main()` calls the function `intfrac()`. Here's some sample interaction:

```
Enter a real number: 99.44
Integer part is 99, fractional part is 0.44
```

Some compilers may generate spurious digits in the fractional part, such as 0.440002. This is an error in the compiler's conversion routine and can be ignored. Refer to Figure 5.7 in the following discussion.

The `intfrac()` function finds the integer part by converting the number (which was passed to the parameter `n`) into a variable of type `long` with a cast, using the expression

```
long temp = static_cast<long>(n);
```

This effectively chops off the fractional part of the number, since integer types (of course) store only the integer part. The result is then converted back to type `float` with another cast:

```
intp = static_cast<float>(temp);
```

The fractional part is simply the original number less the integer part. (We should note that a library function, `fmod()`, performs a similar task for type `double`.)

The `intfrac()` function can find the integer and fractional parts, but how does it pass them back to `main()`? It could use a return statement to return one value, but not both. The problem is solved using reference arguments. Here's the declarator for the function:

```
void intfrac(float n, float& intp, float& fracp)
```

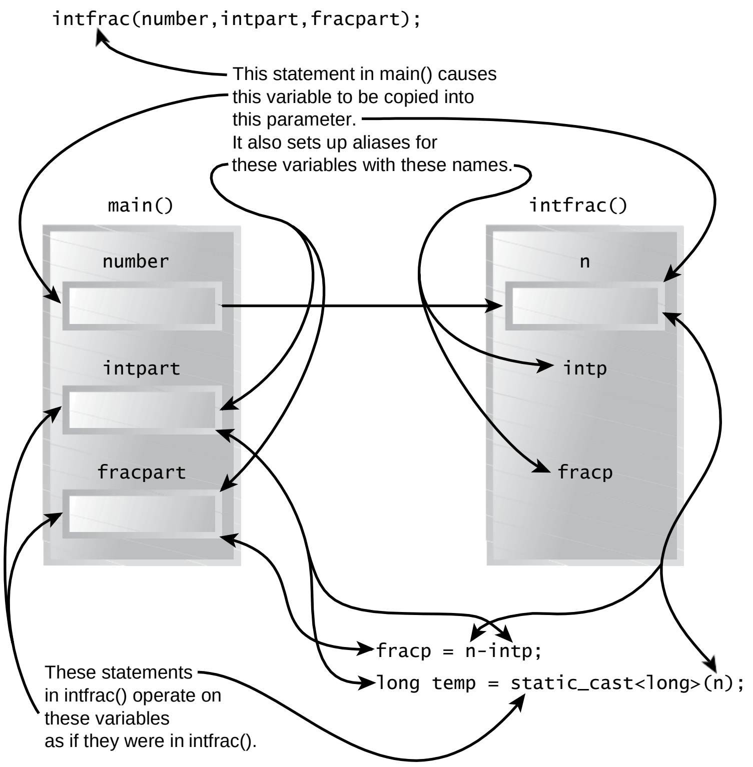

intfrac(number,intpart,fracpart);
This statement in main() causes this variable to be copied into this parameter.
It also sets up aliases for these variables with these names.
main()
number
intpart
fracpart
intfrac()
n
intp
fracp
fracp = n-intp;
long temp = static_cast<long>(n);
These statements in intfrac() operate on these variables as if they were in intfrac().

FIGURE 5.7
Passing by reference in the REF program.

Reference arguments are indicated by the ampersand (&) following the data type:

```
float& intp
```

The & indicates that `intp` is an *alias*—another name—for whatever variable is passed as an argument. In other words, when you use the name `intp` in the `intfrac()` function, you are really referring to `intpart` in `main()`. The & can be taken to mean *reference to*, so

```
float& intp
```

means `intp` is a reference to the `float` variable passed to it. Similarly, `fracp` is an alias for—or a reference to—`fracpart`.

The function declaration echoes the usage of the ampersand in the definition:

```
void intfrac(float, float&, float&);   // ampersands
```

As in the definition, the ampersand follows those arguments that are passed by reference.

The ampersand is not used in the function call:

```
intfrac(number, intpart, fracpart);    // no ampersands
```

From the function call alone, there's no way to tell whether an argument will be passed by reference or by value.

While `intpart` and `fracpart` are passed by reference, the variable `number` is passed by value. `intp` and `intpart` are different names for the same place in memory, as are `fracp` and `fracpart`. On the other hand, since it is passed by value, the parameter `n` in `intfrac()` is a separate variable into which the value of `number` is copied. It can be passed by value because the `intfrac()` function doesn't need to modify `number`.

(C programmers should not confuse the ampersand that is used to mean `reference to` with the same symbol used to mean `address of`. These are different usages. We'll discuss the *address of* meaning of `&` in Chapter 10, "Pointers.")

A More Complex Pass by Reference

Here's a somewhat more complex example of passing simple arguments by reference. Suppose you have pairs of numbers in your program and you want to be sure that the smaller one always precedes the larger one. To do this you call a function, `order()`, which checks two numbers passed to it by reference and swaps the originals if the first is larger than the second. Here's the listing for REFORDER:

```
// reforder.cpp
// orders two arguments passed by reference
#include <iostream>
using namespace std;

int main()
   {
   void order(int&, int&);          //prototype

   int n1=99, n2=11;                //this pair not ordered
   int n3=22, n4=88;                //this pair ordered

   order(n1, n2);                   //order each pair of numbers
   order(n3, n4);

   cout << "n1=" << n1 << endl;     //print out all numbers
   cout << "n2=" << n2 << endl;
   cout << "n3=" << n3 << endl;
   cout << "n4=" << n4 << endl;
   return 0;
   }
```