Function Return Values and Prototypes

So far, you have passed variables to functions in only one direction—a calling function passed data to a receiving function. You have yet to see how data are passed back *from* the receiving function to the calling function. When you pass variables by address, the data are changed in both functions—but this is different from passing data back. This chapter focuses on writing function return values that improve your programming power.

After you learn to pass and return values, you have to *prototype* your own functions as well as C++'s built-in functions, such as `cout` and `cin`. By prototyping your functions, you ensure the accuracy of passed and returned values.

This chapter introduces you to the following:

- Returning values from functions
- Prototyping functions
- Understanding header files

By returning values from functions, you make your functions fully modular. They can now stand apart from the other functions.

They can receive and return values and act as building blocks that compose your complete application.

Function Return Values

Until now, all functions in this book have been *subroutines* or *subfunctions*. A C++ subroutine is a function that is called from another function, but it does not return any values. The difference between subroutines and functions is not as critical in C++ as it is in other languages. All functions, whether they are subroutines or functions that return values, are defined in the same way. You can pass variables to each of them, as you have seen throughout this section of the book.

Put the return value at the end of the `return` statement.

Functions that return values offer you a new approach to programming. In addition to passing data one-way, from calling to receiving function, you can pass data back from a receiving function to its calling function. When you want to return a value from a function to its calling function, put the return value after the `return` statement. To clarify the return value even more, many programmers put parentheses around the return value, as shown in the following syntax:

```
return (return value);
```

CAUTION: Do not return global variables. There is no need to do so because their values are already known throughout the code.

The calling function must have a use for the return value. For example, suppose you wrote a function that calculated the average of any three integer variables passed to it. If you return the average, the calling function has to receive that return value. The following sample program helps to illustrate this principle.


```
// Filename: C19AVG.CPP
// Calculates the average of three input values.
#include <iostream.h>
int calc_av(int num1, int num2, int num3);   //Prototype
```

```
main()
{
   int num1, num2, num3;
   int avg;                  // Holds the return value.

   cout << "Please type three numbers (such as 23 54 85) ";
   cin >> num1 >> num2 >> num3;

   // Call the function, pass the numbers,
   // and accept the return value amount.
   avg = calc_av(num1, num2, num3);

   cout << "\n\nThe average is " << avg;      // Print the
                                              // return value.
   return 0;
}

int calc_av(int num1, int num2, int num3)
{
   int local_avg;  // Holds the average for these numbers.
   local_avg = (num1+num2+num3) / 3;

   return (local_avg);
}
```

Here is a sample output from the program:

```
Please type three numbers (such as 23 54 85) 30 40 50


The average is 40
```

Study this program carefully. It is similar to many you have seen, but a few additional points have to be considered now that the function returns a value. It might help to walk through this program a few lines at a time.

The first part of `main()` is similar to other programs you have seen. It declares its local variables: three for user input and one for the calculated average. The `cout` and `cin` are familiar to you. The function call to `calc_av()` is also familiar; it passes three variables

(num1, num2, and num3) by value to calc_av(). (If it passed them by address, an ampersand (&) would have to precede each argument, as discussed in Chapter 18.)

Put the function's return type before its name. If you don't specify a return type, int is the default.

The receiving function, calc_av(), seems similar to others you have seen. The only difference is that the first line, the function's definition line, has one addition—the int before its name. This is the *type* of the return value. You must always precede a function name with its return data type. If you do not specify a type, C++ assumes a type of int. Therefore, if this example had no return type, it would work just as well because an int return type would be assumed.

Because the variable being returned from calc_av() is an integer, the int return type is placed before calc_av()'s name.

You can see also that the return statement of calc_av() includes the return value, local_avg. This is the variable being sent back to the calling function, main(). You can return only a single variable to a calling function.

Even though a function can receive more than one parameter, it can return only a single value to the calling function. If a receiving function is modifying more than one value from the calling function, you must pass the parameters by address; you cannot return multiple values using a return statement.

After the receiving function, calc_av(), returns the value, main() must do something with that returned value. So far, you have seen function calls on lines by themselves. Notice in main() that the function call appears on the right side of the following assignment statement:

```
avg = calc_av(num1, num2, num3);
```

When the calc_av() function returns its value—the average of the three numbers—that value replaces the function call. If the average computed in calc_av() is 40, the C++ compiler interprets the following statement in place of the function call:

```
avg = 40;
```

You typed a function call to the right of the equal sign, but the program replaces a function call with its return value when the return takes place. In other words, a function that returns a value

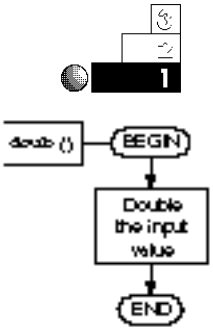
becomes that value. You must put such a function anywhere you put any variable or literal (usually to the right of an equal sign, in an expression, or in `cout`). The following is an *incorrect* way of calling `calc_av()`:

```
calc_av(num1, num2, num3);
```

If you did this, C++ would have nowhere to put the return value.

CAUTION: Function calls that return values usually don't appear on lines by themselves. Because the function call is replaced by the return value, you should do something with that return value (such as assign it to a variable or use it in an expression). Return values can be ignored, but doing so usually defeats the purpose of creating them.

Examples



1. The following program passes a number to a function called `doub()`. The function doubles the number and returns the result.

```
// Filename: C19DOUB.CPP
// Doubles the user's number.
#include <iostream.h>
int doub (int num);
main()
{
   int number;                        // Holds user's input.
   int d_number;     // Holds double the user's input.
   cout << "What number do you want doubled? ";
   cin >> number;

   d_number = doub(number);         // Assigns return value.
   cout << number << " doubled is " << d_number;
   return 0;
}
```

```
int doub(int num)
{
   int d_num;
   d_num = num * 2;                    // Doubles the number.
   return (d_num);                     // Returns the result.
}
```

The program produces output such as this:

```
What number do you want doubled? 5
5 doubled is 10
```

2. Function return values can be used *anywhere* literals, variables, and expressions are used. The following program is similar to the previous one. The difference is in `main()`.

   The function call is performed not on a line by itself, but from a `cout`. This is a nested function call. You call the built-in function `cout` using the return value from one of the program's functions named `doub()`. Because the call to `doub()` is replaced by its return value, the `cout` has enough information to proceed as soon as `doub()` returns. This gives `main()` less overhead because it no longer needs a variable called `d_number`, although you must use your own judgment as to whether this program is easier to maintain. Sometimes it is wise to include function calls in other expressions; other times it is clearer to call the function and assign its return value to a variable before using it.

```
// Filename: C19DOUB2.CPP
// Doubles the user's number.
#include <iostream.h>
int doub(int num); // Prototype

main()
{
   int number;                         // Holds user's input.
   cout << "What number do you want doubled? ";
   cin >> number;
```

```
   // The third cout parameter is
   // replaced with a return value.
   cout << number << " doubled is " << doub(number);

   return 0;
}

int doub(int num)
{
   int d_num;
   d_num = num * 2;                    // Double the number.
   return (d_num);                     // Return the result.
}
```

3. The following program asks the user for a number. That number is then passed to a function called `sum()`, which adds the numbers from `1` to that number. In other words, if the user types a `6`, the function returns the result of the following calculation:

```
1 + 2 + 3 + 4 + 5 + 6
```

This is known as the *sum of the digits* calculation, and it is sometimes used for depreciation in accounting.

```
// Filename: C19SUMD.CPP
// Compute the sum of the digits.
#include <iostream.h>
int sum(int num);  // Prototype

main()
{
   int num, sumd;

   cout << "Please type a number: ";
   cin >> num;

   sumd = sum(num);
   cout << "The sum of the digits is " << sumd;
   return 0;
}
```

```
int sum(int num)
{
   int ctr;                          // Local loop counter.
   int sumd=0;                    // Local to this function.
   if (num <= 0)  // Check whether parameter is too small.
     { sumd = num; }    // Returns parameter if too small.
   else
     { for (ctr=1; ctr<=num; ctr++)
          { sumd += ctr; }
     }
   return(sumd);
}
```

The following is a sample output from this program:

```
Please type a number: 6
The sum of the digits is 21
```

4. The following program contains two functions that return values. The first function, `maximum()`, returns the larger of two numbers entered by the user. The second one, `minimum()`, returns the smaller.

```
// Filename: C19MINMX.CPP
// Finds minimum and maximum values in functions.
#include <iostream.h>

int maximum(int num1, int num2);  // Prototypes
int minimum(int num1, int num2);

main()
{
   int num1, num2;                  // User's two numbers.
   int min, max;

   cout << "Please type two numbers (such as 46 75) ";
   cin >> num1 >> num2;

   max = maximum(num1, num2);      // Assign the return
   min = minimum(num1, num2);      // value of each
                                   // function to variables.
```

```
    cout << "The minimum number is " << min << "\n";
    cout << "The maximum number is " << max << "\n";
    return 0;
}

int maximum(int num1, int num2)
{
    int max;                // Local to this function only.
    max = (num1 > num2) ? (num1) : (num2);
    return (max);
}

int minimum(int num1, int num2)
{
    int min;                // Local to this function only.
    min = (num1 < num2) ? (num1) : (num2);
    return (min);
}
```

Here is a sample output from this program:

```
Please type two numbers (such as 46 75) 72 55
The minimum number is 55
The maximum number is 72
```

If the user types the same number, `minimum` and `maximum` are the same.

These two functions can be passed any two integer values. In such a simple example as this one, the user certainly already knows which number is lower or higher. The point of such an example is to show how to code return values. You might want to use similar functions in a more useful application, such as finding the highest paid employee from a payroll disk file.

Function Prototypes

The word *prototype* is sometimes defined as a model. In C++, a function prototype models the actual function. Before completing

your study of functions, parameters, and return values, you must understand how to prototype each function in your program.

C++ requires that you prototype all functions in your program. When prototyping, you inform C++ of the function's parameter types and its return value, if any.

To prototype a function, copy the function's definition line to the top of your program (immediately before or after the `#include <iostream.h>` line). Place a semicolon at the end of the function definition line, and you have the prototype. The definition line (the function's first line) contains the return type, the function name, and the type of each argument, so the function prototype serves as a model of the function that follows.

If a function does not return a value, or if that function has no arguments passed to it, you should still prototype it. Place the keyword `void` in place of the return type or the parameters. `main()` is the only function that you do not have to prototype because it is *self-prototyping;* meaning `main()` is not called by another function. The first time `main()` appears in your program (assuming you follow the standard approach and make `main()` your program's first function), it is executed.

If a function returns nothing, `void` must be its return type. Put `void` in the argument parentheses of function prototypes with no arguments. All functions must match their prototypes.

C++ assumes functions return `int` unless you put a different data return type, or use the `void` keyword.

All `main()` functions in this book have returned a `0`. Why? You now know enough to answer that question. Because `main()` is self-prototyping, and because the `void` keyword never appeared before `main()` in these programs, C++ assumed an `int` return type. All C++ functions prototyped as returning `int` or those without any return data type prototype assume `int`. If you wanted to not put `return 0;` at the end of `main()`'s functions, you must insert `void` before `main()` as in:

```
void main()     // main() self-prototypes to return nothing.
```

You can look at a statement and tell whether it is a prototype or a function definition (the function's first line) by the semicolon on the end. All prototypes, unless you make `main()` self-prototype, end with a semicolon.

Prototype for Safety

Prototyping protects you from programming mistakes. Suppose you write a function that expects two arguments: an integer followed by a floating-point value. Here is the first line of such a function:

```
my_fun(int num, float amount)
```

What if you passed incorrect data types to `my_fun()`? If you were to call this function by passing it two literals, a floating-point followed by an integer, as in

```
my_fun(23.43, 5);          // Call the my_fun() function.
```

the function would not receive correct parameters. It is expecting an integer followed by a floating-point, but you did the opposite and sent it a floating-point followed by an integer.

Prototyping protects your programs from function programming errors.

In regular C programs, mismatched arguments such as these generate no error message even though the data are not passed correctly. C++ requires prototypes so you cannot send the wrong data types to a function (or expect the wrong data type to be returned). Prototyping the previous function results in this:

```
void my_fun(int num, float amount);        // Prototype
```

In doing so, you tell the compiler to check this function for accuracy. You inform the compiler to expect nothing after the `return` statement, not even `0`, (due to the `void` keyword) and to expect an integer followed by a floating-point in the parentheses.

If you break any of the prototype's rules, the compiler informs you of the problem and you can correct it.

Prototype All Functions

You should prototype every function in your program. As just described, the prototype defines (for the rest of the program) which functions follow, their return types, and their parameter types. You should prototype C++'s built-in functions also, such as `printf()` and `scanf()` if you use them.

Think about how you prototype printf(). You don't always pass it the same types of parameters because you print different data with each printf(). Prototyping functions you write is easy: The prototype is basically the first line in the function. Prototyping functions you do not write might seem difficult, but it isn't—you have already done it with every program in this book!

Header files contain built-in function prototypes.

The designers of C++ realized that all functions have to be prototyped. They realized also that you cannot prototype built-in functions, so they did it for you and placed the prototypes in header files on your disk. You have been including the printf() and scanf() prototypes in each program that used them in this book with the following statement:

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

Inside the stdio.h file is a prototype of many of C++'s input and output functions. By having prototypes of these functions, you ensure that they cannot be passed bad values. If someone attempts to pass incorrect values, C++ catches the problem.

Because printf() and scanf() are not used very often in C++, the cout and cin operators have their own header file called iostream.h that you have seen included in this book's programs as well. The iostream.h file does not actually include prototypes for cout and cin because they are operators and not functions, but iostream.h does include some needed definitions to make cout and cin work.

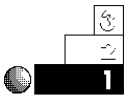
Remember too that iomanip.h has to be included if you use a setw or setprecision modifier in cout. Any time you use a new built-in C++ function or a manipulating operator, check your compiler's manual to find the name of the prototype file to include.

Prototyping is the primary reason why you should always include the matching header file when you use C++'s built-in functions. The strcpy() function you saw in previous chapters requires the following line:

```
#include <string.h>
```

This is the header file for the strcpy() function. Without it, the program does not work.

Examples

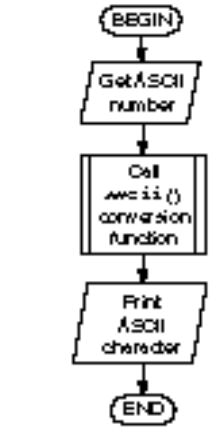


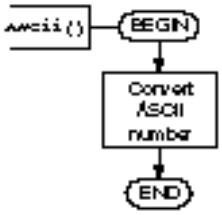
1. Prototype all functions in all programs except `main()`. Even `main()` must be prototyped if it returns nothing (not even `0`). The following program includes two prototypes: one for `main()` because it returns nothing, and one for the built-in `printf()` and `scanf()` functions.

```
// Filename: C19PRO1.CPP
// Calculates sales tax on a sale
#include <stdio.h>        // Prototype built-in functions.
void main(void);

void main(void)
{

   float total_sale;
   float tax_rate = .07;            // Assume seven percent
                                    // tax rate.

   printf("What is the sale amount? ");
   scanf(" %f", &total_sale);

   total_sale += (tax_rate * total_sale);
   printf("The total sale is %.2f", total_sale);
   return;      // No 0 required!
}
```

Notice that `main()`'s `return` statement needed only a semicolon after it. As long as you prototype `main()` with a `void` return type, the last line in `main()` can be `return;` instead of having to type `return 0;` each time.

2. The following program asks the user for a number in `main()`, and passes that number to `ascii()`. The `ascii()` function returns the ASCII character that matches the user's number. This example illustrates a `character` return type. Functions can return any data type.

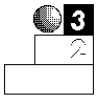





```
// Filename: C19ASC.CPP
// Prints the ASCII character of the user's number.
// Prototypes follow.
#include <iostream.h>
char ascii(int num);

void main()
{
   int num;
   char asc_char;

   cout << "Enter an ASCII number? ";
   cin >> num;

   asc_char = ascii(num);
   cout << "The ASCII character for " << num
        << " is " << asc_char;
   return;
}

char ascii(int num)
{
   char asc_char;
   asc_char = char(num);  // Type cast to a character.
   return (asc_char);
}
```

The output from this program follows:

```
Enter an ASCII number? 67
The ASCII character for 67 is C
```

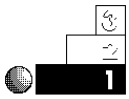


3. Suppose you have to calculate net pay for a company. You find yourself multiplying the hours worked by the hourly pay, then deducting taxes to compute the net pay. The following program includes a function that does this for you. It requires three arguments: the hours worked, the hourly pay, and the tax rate (as a floating-point decimal, such as .30 for 30 percent). The function returns the net pay. The `main()` calling program tests the function by sending three different payroll values to the function and printing the three return values.

```
// Filename: C19NPAY.CPP
// Defines a function that computes net pay.
#include <iostream.h>  // Needed for cout and cin.
void main(void);
float netpayfun(float hours, float rate, float taxrate);

void main(void)
{
   float net_pay;

   net_pay = netpayfun(40.0, 3.50, .20);
   cout << "The pay for 40 hours at $3.50/hr., and a 20% "
        << "tax rate is $";
   cout << net_pay << "\n";

   net_pay = netpayfun(50.0, 10.00, .30);
   cout << "The pay for 50 hours at $10.00/hr., and a 30% "
        << "tax rate is $";
   cout << net_pay << "\n";

   net_pay = netpayfun(10.0, 5.00, .10);
   cout << "The pay for 10 hours at $5.00/hr., and a 10% "
        << " tax rate is $";
   cout << net_pay << "\n";

   return;
}

float netpayfun(float hours, float rate, float taxrate)
{
   float gross_pay, taxes, net_pay;
   gross_pay = (hours * rate);
   taxes = (taxrate * gross_pay);
   net_pay = (gross_pay - taxes);
   return (net_pay);
}
```

Review Questions

The answers to the review questions are in Appendix B.



1. How do you declare function return types?
2. What is the maximum number of return values a function can return?
3. What are header files for?

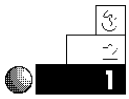
4. What is the default function return type?
5. True or false: a function that returns a value can be passed only a single parameter.

6. How do prototypes protect the programmer from bugs?
7. Why don't you have to return global variables?
8. What is the return type, given the following function prototype?

```
float my_fun(char a, int b, float c);
```

How many parameters are passed to `my_fun()`? What are their types?

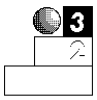
Review Exercises



1. Write a program that contains two functions. The first function returns the square of the integer passed to it, and the second function returns the cube. Prototype `main()` so you do not have to return a value.

2. Write a function that returns the double-precision area of a circle, given that a double-precision radius is passed to it. The formula for calculating the area of a circle is

```
area = 3.14159 * (radius * radius)
```



3. Write a function that returns the value of a polynomial given this formula:

```
9x4 + 15x2 + x1
```

Assume `x` is passed from `main()` and it is supplied by the user.

Summary

You learned how to build your own collection of functions. When you write a function, you might want to use it in more than one program—there is no need to reinvent the wheel. Many programmers write useful functions and use them in more than one program.

You now understand the importance of prototyping functions. You should prototype all your own functions, and include the appropriate header file when you use one of C++'s built-in functions. Furthermore, when a function returns a value other than an integer, you must prototype so C++ recognizes the noninteger return value.