***PD: Please choose View, Print Layout to see the formula in this chapter. AU***

CHAPTER 5

Functions

Objectives

• To create functions, invoke functions, and pass arguments to a function (§5.2-5.4).

• To understand the differences between pass by value and pass by reference (§5.5-5.6).

• To use function overloading and understand ambiguous overloading (§5.7).

• To use function prototypes for declaring function headers (§5.8).

• To know how to use default arguments (§5.13).

• To create header files for reusing functions (§5.10).

• To determine the scope of local and global variables (§5.12).

• To develop applications using the C++ mathematical functions (§5.14).

• To design and implement functions using stepwise refinement (§5.15).
5.1 Introduction

In the preceding chapters, you learned about such functions as pow(a, b), rand(), srand(seed), and main(). A function is a collection of statements that are grouped together to perform an operation. When you call the pow(a, b) function, for example, the system actually executes several statements in order to return the result.

This chapter introduces several topics that involve, or are related to, functions. You will learn how to create your own functions with or without return values, invoke a function with or without parameters, overload functions using the same names, and apply function abstraction in the program design.

5.2 Creating a Function

In general, a function has the following syntax:

```
returnValueTyp functionNam(list of parameters)
{
    // Function body;
}
```

Let’s take a look at a function created to find which of two integers is bigger. This function, named max, has two int parameters, num1 and num2, the larger of which is returned by the function. Figure 5.1 illustrates the components of this function.

***Same as Fig 4.1 in intro5e p126

```c
int max(int num1, int num2)
{
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
```

Figure 5.1
A function declaration consists of a function header and a function body.

<Side Remark: function header>
The function header specifies the return value type, function name, and parameters of the function.

<Side Remark: nonvoid function>

A function may return a value. The returnType is the data type of the value the function returns. Some functions perform desired operations without returning a value. In this case, the
returnValueType is the keyword void. For example, the returnType in the srand function is void. The function that returns a value is called a nonvoid function and the function that does not return a value is called a void function.

*<Side Remark: parameter>*
*<Side Remark: argument>*
*<Side Remark: parameter list>*
*<Side Remark: function signature>*

The variables defined in the function header are known as formal parameters or simply parameters. A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. The function name and the parameter list together constitute the function signature. Parameters are optional; that is, a function may contain no parameters.

The function body contains a collection of statements that define what the function does. The function body of the max function uses an if statement to determine which number is larger and return the value of that number. A return statement using the keyword return is required for a nonvoid function to return a result. The function terminates when a return statement is executed.

**CAUTION**

You need to declare a separate data type for each parameter. For instance, int num1, num2 should be replaced by int num1, int num2.

### 5.3 Calling a Function

In creating a function, you give a definition of what the function is to do. To use a function, you have to call or invoke it. There are two ways to call a function; the choice is based on whether the function returns a value or not.

If the function returns a value, a call to the function is usually treated as a value. For example,

```cpp
int larger = max(3, 4);
```

calls max(3, 4) and assigns the result of the function to the variable larger. Another example of a call that is treated as a value is

```cpp
cout << max(3, 4);
```

which prints the return value of the function call max(3, 4).

**NOTE**
A function with a nonvoid return value type can also be invoked as a statement in C++. In this case, the caller simply ignores the return value. This is rare, but permissible if the caller is not interested in the return value.

When a program calls a function, program control is transferred to the called function. A called function returns control to the caller when its return statement is executed or when its function-ending closing brace is reached.

Listing 5.1 shows a complete program that is used to test the max function.

```cpp
#include <iostream>
using namespace std;

// Return the max between two numbers
int max(int num1, int num2) {
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}

int main() {
    int i = 5;
    int j = 2;
    int k = max(i, j);
    cout << "The maximum between " << i << " and " << j << " is " << k;
    return 0;
}
```

The maximum between 5 and 2 is 5

This program contains the max function and the main function. The main function is just like any other function except that it is invoked by the operating system to execute the program.

A function must be declared before it is invoked. Since the max function is invoked by the main function, it must be declared before the main function.
When the `max` function is invoked (line 20), variable `i`’s value 5 is passed to `num1`, and variable `j`’s value 2 is passed to `num2` in the `max` function. The flow of control transfers to the `max` function. The `max` function is executed. When the `return` statement in the `max` function is executed, the `max` function returns the control to its caller (in this case the caller is the `main` function). This process is illustrated in Figure 5.2.

```c++
int main()
{
    int i = 5;
    int j = 2;
    int k = max(i, j);
    cout << "The maximum between " << i << " and " << j << " is " << k;
    return 0;
}

int max(int num1, int num2)
{
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
```

**Figure 5.2**

When the `max` function is invoked, the flow of control transfers to the `max` function. Once the `max` function is finished, it returns the control back to the caller.

5.3.1 Call Stacks

*Side Remark: stack*

Each time a function is invoked, the system stores parameters and variables in an area of memory, known as a stack, which stores elements in last-in first-out fashion. When a function calls another function, the caller’s stack space is kept intact, and new space is created to handle the new function call. When a function finishes its work and returns to its caller, its associated space is released.

Understanding call stacks helps comprehend how functions are invoked. The variables defined in the `main` function are `i`, `j`, and `k`. The variables defined in the `max` function are `num1`, `num2`, and `result`. The variables `num1` and `num2` are defined in the function signature and are parameters of the function. Their values are passed through function invocation. Figure 5.3 illustrates the variables in the stack.
The main function is invoked.

Space required for the main function
k: 2
j: 5

(c) The max function is finished and the return value is sent to k.

Space required for the main function
k: 5
j: 2
i: 5

(d) The main function is finished.

Figure 5.3

When the max function is invoked, the flow of control transfers to the max function. Once the max function is finished, it returns the control back to the caller.

TIP
<side remark: debugging in IDE>
If you use an IDE such as C++Builder and Visual C++.NET, please refer to Learning C++ Effectively with C++Builder/Visual C++.NET in the supplements. This supplement shows you how to use a debugger to trace function invocations.

5.4 void Functions

The preceding section gives an example of a nonvoid function. This section shows how to declare and invoke a void function. Listing 5.2 gives a program that declares a function named printGrade and invokes it to print the grade for a given score.

Listing 5.2 TestVoidFunction.cpp (Using a void Function)

```cpp
#include <iostream>
using namespace std;

// Print grade for the score
void printGrade(double score)
{
   if (score >= 90.0)
      cout << 'A';
   else if (score >= 80.0)
      cout << 'B';
   else if (score >= 70.0)
      cout << 'C';
   else if (score >= 60.0)
      cout << 'D';
}
```

cout << 'D';
else
  cout << 'F';
}

int main()
{
  cout << "Enter a score: ";
  double score;
  cin >> score;
  cout << "The grade is ";
  printGrade(score);
  return 0;
}

Enter a score: 78.5
The grade is C

<Side Remark: invoke void function>
The printGrade function is a void function. It does not return any value. A call to a void function must be a statement. So, it is invoked as a statement in line 26 in the main function. This statement is like any C++ statement terminated with a semicolon.

NOTE

<subtitle: return in void function>
A return statement is not needed for a void function, but it can be used for terminating the function and returning to the function’s caller. The syntax is simply

return;

This is rare, but sometimes useful for circumventing the normal flow of control in a void function. For example, the following code has a return statement to terminate the function when the score is invalid.

// Print grade for the score
void printGrade(double score)
{
  if (score < 0 || score > 100) {
    cout << "Invalid score";
    return;
  }
  if (score >= 90.0)
    cout << 'A';
  else if (score >= 80.0)
    cout << 'B';
  else if (score >= 70.0)
    cout << 'C';
else if (score >= 60.0)
    cout << 'D';
else
    cout << 'F';

***End NOTE

.Side Remark: parameter order association>
The power of a function is its ability to work with parameters. You can use max to find the maximum between any two int values. When calling a function, you need to provide arguments, which must be given in the same order as their respective parameters in the function specification. This is known as parameter order association. For example, the following function prints a character n times:

```cpp
void nPrintln(char ch, int n)
{
    for (int i = 0; i < n; i++)
        cout << ch;
}
```

You can use nPrintln('a', 3) to print 'a' three times. The nPrintln('a', 3) statement passes the actual string parameter, 'a', to the parameter, ch; passes 3 to n; and prints 'a' three times. However, the statement nPrintln(3, 'a') would have a different meaning. It passes 3 to ch and 'a' to n.

```cpp
#include <iostream>

using namespace std;

void nPrintln(char ch, int n)
{
    for (int i = 0; i < n; i++)
        cout << ch;
}

int main()
{
    nPrintln('a', 3);
    return 0;
}
```

5.5 Passing Parameters by Values

.Side Remark: pass by value>
When you invoke a function with a parameter, the value of the argument is passed to the parameter. This is referred to as pass by value. If the argument is a variable rather than a literal value, the value of the variable is passed to the parameter. The variable is not affected, regardless of the changes made to the parameter inside the function. We will examine an interesting scenario in the following example, in which the parameters are changed in the function but the arguments are not affected.

Listing 5.3 is a program that demonstrates the effect of passing by value. The program creates a function for swapping two variables. The swap function is invoked by passing two arguments. Interestingly, the values of the arguments are not changed after the function is invoked.

Listing 5.3 TestPassByValue.cpp (Passing by Value)
```cpp
#include <iostream>
using namespace std;

// Swap two variables
void swap(int n1, int n2)
{
    cout << "Inside the swap function" << endl;
    cout << "Before swapping n1 is " << n1 << " n2 is " << n2 << endl;

    // Swap n1 with n2
    int temp = n1;
    n1 = n2;
    n2 = temp;

    cout << "After swapping n1 is " << n1 << " n2 is " << n2 << endl;

}

int main()
{
    // Declare and initialize variables
    int num1 = 1;
    int num2 = 2;

    cout << "Before invoking the swap function, num1 is " << num1 << " and num2 is " << num2 << endl;

    // Invoke the swap function to attempt to swap two variables
    swap(num1, num2);

    cout << "After invoking the swap function, num1 is " << num1 << " and num2 is " << num2 << endl;

    return 0;
}
```

**Output**

Before invoking the swap function, num1 is 1 and num2 is 2
Inside the swap function
Before swapping n1 is 1 n2 is 2
After swapping n1 is 2 n2 is 1

After invoking the swap function, num1 is 1 and num2 is 2

**End Output**

Before the swap function is invoked (line 30), num1 is 1 and num2 is 2. After the swap function is invoked, num1 is still 1 and num2 is still 2. Their values are not swapped after the swap function is invoked. As shown in Figure 5.4, the values of the arguments num1 and num2 are passed to n1 and n2, but n1 and n2 have their own memory locations independent of num1 and num2.
num2. Therefore, changes in \( n_1 \) and \( n_2 \) do not affect the contents of \( num1 \) and \( num2 \).

***Same as Fig 4.6 in intro5e p132***

![Diagram showing memory space for main function and swap function]

**Figure 5.4**

The values of the variables are passed to the parameters of the function.

Another twist is to change the parameter name \( n_1 \) in \( swap \) to \( num1 \). What effect does this have? No change occurs because it makes no difference whether the parameter and the argument have the same name. The parameter is a variable in the function with its own memory space. The variable is allocated when the function is invoked, and it disappears when the function is returned to its caller.

**5.6 Passing Arguments by References**

Listing 5.3, TestPassByValue.cpp, presented a \( swap \) function that attempted to swap two variables. But the values of the variables are not swapped after invoking the function, because the values of variables are passed to the arguments. The original variable and arguments are independent. The values in original variables are not changed even though the values in the arguments are changed.

C++ provides a special type of variable, called a **reference variable**, which can be used as a function parameter to reference the original variable. A reference variable is an alias for another variable. Any changes made through the reference variable are actually performed on the original variable. To declare a reference variable, place the ampersand (\&) in front of the name. For example, see Listing 5.4.

**Listing 5.4 TestReferenceVariable.cpp (Using Reference Variable)**

***PD: Please add line numbers in the following code***

**<Side Remark line 7: declare reference variable>**
Line 7 declares a reference variable named refCount that references the original variable count. Line 8 increments refCount. Since refCount is merely an alias for count, count is incremented. So, count is 2 and refCount is also 2.

You can use a reference variable as a parameter in a function and pass a regular variable to invoke the function. The parameter becomes an alias for the original variable. This is known as pass by reference. When you change the value through the reference variable, the original value is actually changed.

Now you can use reference parameters to implement a correct swap function as shown in Listing 5.5.
cout << "\t\tAfter swapping n1 is " << n1 << " n2 is " << n2 << endl;

int main()
{
    // Declare and initialize variables
    int num1 = 1;
    int num2 = 2;

    cout << "Before invoking the swap function, num1 is " << num1 << " and num2 is " << num2 << endl;

    // Invoke the swap function to attempt to swap two variables
    swap(num1, num2);

    cout << "After invoking the swap function, num1 is " << num1 << " and num2 is " << num2 << endl;

    return 0;
}

<Output>
Before invoking the swap function, num1 is 1 and num2 is 2
    Inside the swap function
        Before swapping n1 is 1 n2 is 2
        After swapping n1 is 2 n2 is 1
After invoking the swap function, num1 is 2 and num2 is 1
<End Output>

Before the swap function is invoked (line 30), num1 is 1 and num2 is 2. After the swap function is invoked, num1 becomes 2 and num2 becomes 1. Their values are swapped after the swap function is invoked.

5.7 Overloading Functions

The max function that was used earlier works only with the int data type. But what if you need to find which of two floating-point numbers has the maximum value? The solution is to create another function with the same name but different parameters, as shown in the following code:

    double max(double num1, double num2)
    {
        if (num1 > num2)
            return num1;
        else
            return num2;
    }

<Side Remark: function overloading>
If you call max with int parameters, the max function that expects int parameters will be invoked; if you call max with double parameters, the max function that expects double parameters will be invoked. This is referred to as function overloading; that is, two functions have the same name but
different parameter lists within one file. The C++ compiler determines which function is used based on the function signature.

Listing 5.6 is a program that creates three functions. The first finds the maximum integer, the second finds the maximum double, and the third finds the maximum among three double values. All three functions are named `max`.

```cpp
***PD: Please add line numbers in the following code***
<Side Remark line 5: max function>
<Side Remark line 14: max function>
<Side Remark line 23: max function>
<Side Remark line 28: main function>
<Side Remark line 31: invoke max>
<Side Remark line 35: invoke max>
<Side Remark line 39: invoke max>

#include <iostream>
using namespace std;

// Return the max between two int values
int max(int num1, int num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}

// Find the max between two double values
double max(double num1, double num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}

// Return the max among three double values
double max(double num1, double num2, double num3)
{
    return max(max(num1, num2), num3);
}

int main()
{
    // Invoke the max function with int parameters
    cout << "The maximum between 3 and 4 is " << max(3, 4) << endl;

    // Invoke the max function with the double parameters
    cout << "The maximum between 3.0 and 5.4 is "
        << max(3.0, 5.4) << endl;
```

172
// Invoke the max function with three double parameters
cout << "The maximum between 3.0, 5.4, and 10.14 is "
<< max(3.0, 5.4, 10.14) << endl;
return 0;

When calling \texttt{max(3, 4)} (line 31), the \texttt{max} function for finding the maximum of two integers is invoked. When calling \texttt{max(3.0, 5.4)} (line 35), the \texttt{max} function for finding the maximum of two doubles is invoked. When calling \texttt{max(3.0, 5.4, 10.14)} (line 39), the \texttt{max} function for finding the maximum of three double values is invoked.

Can you invoke the \texttt{max} function with an \texttt{int} value and a \texttt{double} value, such as \texttt{max(2, 2.5)}? If so, which of the \texttt{max} functions is invoked? The answer to the first question is yes. The answer to the second is that the \texttt{max} function for finding the maximum of two \texttt{double} values is invoked. The argument value 2 is automatically converted into a \texttt{double} value and passed to this function.

You may be wondering why the function \texttt{max(double, double)} is not invoked for the call \texttt{max(3, 4)}. Both \texttt{max(double, double)} and \texttt{max(int, int)} are possible matches for \texttt{max(3, 4)}. The C++ compiler finds the most specific function for a function invocation. Since the function \texttt{max(int, int)} is more specific than \texttt{max(double, double)}, \texttt{max(int, int)} is used to invoke \texttt{max(3, 4)}.

\textbf{TIP}

Overloading functions can make programs clearer and more readable. Functions that perform closely related tasks should be given the same name.

\textbf{NOTE}

Overloaded functions must have different parameter lists. You cannot overload functions based on different return types.

\textbf{NOTE}

\textbf{<Side Remark: ambiguous invocation>}

Sometimes there are two or more possible matches for an invocation of a function, but the compiler cannot determine the most specific match. This is referred to as \textit{ambiguous invocation}. Ambiguous invocation causes a compilation error. Consider the following code:
```cpp
#include <iostream>
using namespace std;

int maxNumber(int num1, double num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}

double maxNumber(double num1, int num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}

int main()
{
    cout << maxNumber(1, 2) << endl;
    return 0;
}
```

Both `maxNumber(int, double)` and `maxNumber(double, int)` are possible candidates to match `maxNumber(1, 2)`. Since neither of them is more specific than the other, the invocation is ambiguous, resulting in a compilation error.

If you change `maxNumber(1, 2)` to `maxNumber(1, 2.0)`, it will match the first `maxNumber` function. So, there will be no compilation errors.

***End of NOTE

### 5.8 Function Prototypes

Before a function is called, it must be declared first. One way to ensure it is to place the declaration before all function calls. Another way to approach it is to declare a function prototype before the function is called. A function prototype is a function declaration without implementation. The implementation can be given later in the program.

Listing 5.7 rewrites TestingFunctionOverloading.cpp in Listing 5.4 using function prototypes. Three `max` function prototypes are defined in lines 5-7. These functions are called later in the `main` function. The functions are implemented in lines 27, 36, and 45.

Listing 5.7 TestFunctionPrototype.cpp (Overloading Functions)

***PD: Please add line numbers in the following code***
#include <iostream>
using namespace std;

// Function prototype
int max(int num1, int num2);
double max(double num1, double num2);
double max(double num1, double num2, double num3);

int main()
{
    // Invoke the max function with int parameters
    cout << "The maximum between 3 and 4 is " << 
    max(3, 4) << endl;

    // Invoke the max function with the double parameters
    cout << "The maximum between 3.0 and 5.4 is " 
    << max(3.0, 5.4) << endl;

    // Invoke the max function with three double parameters
    cout << "The maximum between 3.0, 5.4, and 10.14 is " 
    << max(3.0, 5.4, 10.14) << endl;

    return 0;
}

// Return the max between two int values
int max(int num1, int num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}

// Find the max between two double values
double max(double num1, double num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}
// Return the max among three double values
double max(double num1, double num2, double num3)
{
    return max(max(num1, num2), num3);
}

NOTE:

<side remark: omitting parameter names>
It is not necessary to list parameter names in the prototype, only the parameter types are required. C++ compiler ignores the parameter names. The prototype tells the compiler the name of the function, its return type, number of the parameters, and the type of each parameter. So lines 5-7 can be replaced by

    int max(int, int);
    double max(double, double);
    double max(double, double, double);

***End NOTE

5.9 Default Arguments

C++ allows you to declare functions with default argument values. The default values are passed to the parameters when a function is invoked without the arguments.

Listing 5.8 demonstrates how to declare functions with default argument values and how to invoke such functions.

Listing 5.8 DefaultArgumentDemo.cpp (Using Default Arguments)

***PD: Please add line numbers in the following code***
<Side Remark line 5: default argument>
<Side Remark line 13: invoke with default>
<Side Remark line 14: invoke with argument>

```cpp
#include <iostream>
using namespace std;

// Swap two variables
void printArea(double radius = 1)
{
    double area = radius * radius * 3.14159;
    cout << "area is " << area << endl;
}

int main()
{
    printArea();
    printArea(4);
    return 0;
}
```

<Output>
Line 5 declares the `printArea` function with the parameter `radius`. `radius` has a default value 1. Line 13 invokes the function without passing an argument. In this case, the default value 1 is assigned to `radius`.

<side remark: default arguments last>
When a function contains a mixture of parameters with and without default values, the parameters with default values must be defined last. For example, the following declarations are illegal:

```c
void t1(int x, int y = 0, int z);
void t2(int x = 0, int y = 0, int z);
```

However, the following declarations are fine:

```c
void t3(int x, int y = 0, int z = 0);
void t4(int x = 0, int y = 0, int z = 0);
```

When an argument is left out of a function, all arguments that come after it must be left out as well. For example, the following calls are illegal:

```c
t3(1, , 20);
t4(, , 20);
```

But the following calls are fine:

```c
t3(1); // parameters y and z are assigned a default value
t4(1, 2); // parameter z is assigned a default value
```

5.10 Case Study: Computing Taxes with Functions

The program in Listing 3.4, “Computing Taxes,” uses if statements to check the filing status and computes the tax based on the filing status. This example uses functions to simplify Listing 3.4.

Each filing status has six brackets. The code for computing taxes is nearly the same for each filing status except that each filing status has different bracket ranges. For example, the single filer status has six brackets [0, 6000], (6000, 27950], (27950, 67700], (67700, 141250], (141250, 307050], (307050, ∞), and the married file jointly status has six brackets [0, 12000], (12000, 46700], (46700, 112850], (112850, 171950], (171950, 307050], (307050, ∞). The first bracket of each filing status is taxed at 10%, the second at 15%, the third at 27%, the fourth at 30%, the fifth at 35%, and the sixth at 38.6%. So you can write a function with the brackets
as arguments to compute the tax for the filing status. The
header of the function is:

```c++
double computeTax(double income,
                  int r1, int r2, int r3, int r4, int r5)
```

For example, you can invoke `computeTax(400000, 6000, 27950,
67700, 141250, 307050)` to compute the tax for single filers
with $400,000 of taxable income.

Listing 5.9 gives the solution to the problem.

```c++
#include <iostream>
#include <iomanip>
using namespace std;

// Function prototype, parameter names are omitted
double computeTax(int, double); 

int main()
{
    // Prompt the user to enter filing status
    cout << "Enter the filing status
    \t(0-single, 1-joint, 2-separate, 3-head of house hold): ";
    int status;
    cin >> status;

    // Prompt the user to enter taxable income
    cout << "Enter the taxable income: ";
    double income;
    cin >> income;

    // Display the result
    cout << "Tax is " << fixed << setprecision(2) << computeTax(status, income);
    return 0;
}

double computeTax(
    double income, int r1, int r2, int r3, int r4, int r5)
```
double tax = 0;

if (income <= r1)
    tax = income * 0.10;
else if (income <= r2)
    tax = r1 * 0.10 + (income - r1) * 0.15;
else if (income <= r3)
    tax = r1 * 0.10 + (r2 - r1) * 0.15 + (income - r2) * 0.27;
else if (income <= r4)
    tax = r1 * 0.10 + (r2 - r1) * 0.15 + (r3 - r2) * 0.27 +
        (income - r3) * 0.30;
else if (income <= r5)
    tax = r1 * 0.10 + (r2 - r1) * 0.15 + (r3 - r2) * 0.27 +
        (r4 - r3) * 0.30 + (income - r4) * 0.35;
else
    tax = r1 * 0.10 + (r2 - r1) * 0.15 + (r3 - r2) * 0.27 +
        (r4 - r3) * 0.30 + (r5 - r4) * 0.35 + (income - r5) * 0.386;

return tax;

double computeTax(int status, double income)
{
    switch (status)
    {
        case 0:
            return computeTax(income, 6000, 27950, 67700, 141250, 307050);
        case 1:
            return computeTax(income, 12000, 46700, 112850, 171950, 307050);
        case 2:
            return computeTax(income, 6000, 23350, 56425, 85975, 153525);
        case 3:
            return computeTax(income, 10000, 37450, 96700, 156600, 307050);
        default:
            return 0;
    }
}

<Output>
Enter the filing status (0-single, 1-joint, 2-separate, 3-head of house hold): 0
Enter the taxable income: 454542
Tax is 151651.91
</End Output>

This program does the same thing as the one in Listing 3.4. Instead of writing the same code for computing taxes for different filing statuses, the new program uses a function for computing taxes. Using the function not only shortens the program, it also makes the program simpler, easy to read, and easy to maintain.

The program uses two overloaded computeTax functions (lines 28, 52). The first computeTax function in line 28 computes the tax for the specified brackets and taxable income. The second
computeTax function in line 52 computes the tax for the specified status and taxable income. Because the first computeTax function is invoked by the second, the first is declared before the second one. Because the second computeTax function is invoked from the main function, its function prototype is declared before the main function in line 6. The parameter names in the function prototype are omitted.

### 5.11 Reusing Functions by Different Programs

One of the benefits of functions is for reuse. In the preceding sections, you declared functions and used them from the same program. To make the functions available for other programs to use, you need to place the functions in a separate file, called header file. By convention, the file has a .h extension. Programs use `#include` preprocessor directives to include header files in order to reuse the functions defined in the header file.

Listing 5.8 creates a header file named MyLib.h. This file declares a function named `isEven(number)` that returns `true` if the number is even.

Listing 5.8 MyLib.h (Header File Demo)

```cpp
***PD: Please add line numbers in the following code***

bool isEven(int number) 
{ 
    return (number % 2 == 0); 
} 
```

Listing 5.10 creates a file named TestMyLib.cpp. This file contains a main function for testing the `isEven` function.

Listing 5.10 TestMyLib.cpp (Using MyLib)

```cpp
***PD: Please add line numbers in the following code***

#include <iostream>
#include "MyLib.h"
using namespace std;

int main() 
{ 
    cout << isEven(4) << endl; 
    cout << isEven(5) << endl; 
    return 0; 
} 
```

<Output>

```cpp
1
0
```
The program includes two header files iostream (line 1) and MyLib.h (line 2). iostream is C++ standard header file and MyLib.h is a user-defined header file. A user-defined header file should be enclosed in double quotation marks (" ") and a standard header file should be enclosed angle brackets (< >).

Header files can be placed anywhere on the disk. For example, if MyLib.h is placed under the c:\ root directory on Windows, you can include it using the absolute file name as follows:

```
#include "c:\MyLib.h"
```

This makes your code dependent on Windows. To fix the problem, place the header in or under the same directory with the program that includes the header file.

CAUTION

Header files are designed for reuse by other programs that likely have a main function. Because one program cannot have two main functions, you should not write a main function in header files.

5.12 Case Study: Generating Random Characters

Computer programs process numerical data and characters. You have seen many examples that involve numerical data. It is also important to understand characters and how to process them. This section presents an example for generating random characters.

As introduced in §2.11, every character has a unique ASCII code between 0 and 127. To generate a random character is to generate a random integer between 0 and 127. You learned how to generate a random number in §3.8. Recall that you can use the `srand(seed)` function to set a seed and use `rand()` to return a random integer. You can use it to write a simple expression to generate random numbers in any range. For example,

```
random() % 10  \quad \Rightarrow \quad \text{Returns a random integer between 0 and 9.}
```

```
50 + rand() % 50  \quad \Rightarrow \quad \text{Returns a random integer between 50 and 99.}
```
In general,\[ a + \text{rand()} \% b \]

Returns a random number between \(a\) and \(a + b\), excluding \(a + b\).

So, you can use the following expression to generate a random integer between 0 and 127:

\[
\text{rand()} \% 128
\]

Now let us consider how to generate a random lowercase letter. The ASCII for lowercase letters are consecutive integers starting from the code for 'a', then for 'b', 'c', ..., and 'z'. The code for 'a' is \(\text{static_cast<int>__('a')}\).

So a random integer between \(\text{static_cast<int>__('a')}\) and \(\text{static_cast<int>__('z')}\) is

\[
\text{static_cast<int>__('a')} + \text{rand()} \% (\text{static_cast<int>__('z')} - \text{static_cast<int>__('a')} + 1)
\]

As discussed in §2.11.2, all numeric operators can be applied to the char operands. The char operand is cast into a number if the other operand is a number or a character. Thus the preceding expression can be simplified as follows:

\[
\text{'a'} + \text{rand()} \% (\text{'z'} - \text{'a'} + 1)
\]

and a random lowercase letter is

\[
\text{static_cast<char>__('a') + rand() \% ('z' - 'a' + 1)}
\]

To generalize the foregoing discussion, a random character between any two characters \(\text{ch1}\) and \(\text{ch2}\) with \(\text{ch1} < \text{ch2}\) can be generated as follows:

\[
\text{static_cast<char>(ch1 + rand() \% (ch2 – ch1 + 1))}
\]

This is a simple but useful discovery. Let us create a header file named RandomChacter.h in Listing 5.12 with five overloaded functions to get a certain type of character randomly. You can use these functions in your future projects.

**Listing 5.12 RandomCharacter.h (Generating Random Characters)**

```cpp
#include <cstdlib>
using namespace std;

// Generate a random character between ch1 and ch2
char getRandomCharacter(char ch1, char ch2)
{
    return static_cast<char>(ch1 + rand() % (ch2 – ch1 + 1));
}

// Generate a random lowercase letter
char getRandomLowerCaseLetter()
{
    return getRandomCharacter('a', 'z');
}

// Generate a random uppercase letter
char getRandomUpperCaseLetter()
{
    return getRandomCharacter('A', 'Z');
}
```

**PD: Please add line numbers in the following code***
return getRandomCharacter('A', 'Z');
}

// Generate a random digit character
char getRandomDigitCharacter()
{
    return getRandomCharacter('0', '9');
}

// Generate a random character
char getRandomCharacter()
{
    return getRandomCharacter(0, 127);
}

Listing 5.13 gives a test program that displays one hundred lowercase letters.

Listing 5.13 TestRandomCharacter.cpp (Using RandomCharacter)

***PD: Please add line numbers in the following code***

<Side Remark line 7: constants>
<Side Remark line 10: new seed>
<Side Remark line 15: lower-case letter>

```cpp
#include <iostream>
#include "RandomCharacter.h"
using namespace std;

int main()
{
    const int NUMBER_OF_CHARS = 175;
    const int CHARS_PER_LINE = 25;
    srand(time(0)); // Set a new seed for random function

    // Print random characters between '!' and '~', 25 chars per line
    for (int i = 0; i < NUMBER_OF_CHARS; i++)
    {
        char ch = getRandomLowerCaseLetter();
        if ((i + 1) % CHARS_PER_LINE == 0)
            cout << ch << endl;
        else
            cout << ch;
    }

    return 0;
}
```

<Output>

gmjsohezfkgtasqgmswflaor
porunuiwmastifjedmpchot
isqgtvxxxbzuirmqmhitr
lbrrlisopfxahsqhwwuujvbe
xbhdotzhehboqmmwfsfktswol1
cbuwkgxqmtzhigatsl1vbwbz
bfesoklwbnnooyg11gdxugq1

<End Output>

Line 2 includes RandomCharacter.h, since the program invokes the function defined in this header file.

The getRandomLowerCaseLetter() function utilizes the rand() function to obtain a random character. To ensure that you get a sequence of different random numbers, srand(time(0)) is invoked in line 10 to set a new seed for the random number generator algorithm.

<side remark: parentheses required>
Note that `getRandomLowerCaseLetter()` does not have any parameters, but you still have to use the parentheses when defining and invoking the function.

### 5.13 The Scope of Variables

The scope of a variable is the part of the program where the variable can be referenced. A variable defined inside a function is referred to as a `local variable`. C++ also allows you to use `global variables`. They are declared outside all functions and are accessible to all functions in its scope. Local variables do not have default values, but global variables are defaulted to zero.

A variable must be declared before it can be used. The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. The scope of a global variable starts from its declaration and continues to the end of the program.

A parameter is actually a local variable. The scope of a function parameter covers the entire function.

Listing 5.14 demonstrates the scope of local and global variables.

```cpp
#include <iostream>
using namespace std;

void t1(); // function prototype
void t2(); // function prototype

int main()
{
    t1();
    t2();
    return 0;
}

int y; // Global variable, default to 0
```
void t1()
{
    int x = 1;
    cout << "x is " << x << endl;
    cout << "y is " << y << endl;
    x++;
    y++;
}

void t2()
{
    int x = 1;
    cout << "x is " << x << endl;
    cout << "y is " << y << endl;
}

<Output>
x is 1
y is 1
x is 2
y is 1

<End Output>

A global variable \( y \) is declared in line 15 with default value 0. This variable is accessible in functions \( t1 \) and \( t2 \), but not in the main function, because the main function is declared before \( y \) is declared.

When the main function invokes \( t1() \) in line 9, the global variable \( y \) is incremented (line 23) and becomes 1 in \( t1 \). When the main function invokes \( t2() \) in line 10, the global variable \( y \) is now 1.

A local variable \( x \) is declared in \( t1 \) in line 19 and another local variable \( x \) is declared in \( t2 \) in line 28. These two variables are independent, although they named the same. So incrementing \( x \) in \( t1 \) does not effect the variable \( x \) defined in \( t2 \).

If a function has a local variable with the same name as a global variable, only the local variable can be seen from the function.

TIP:
<side remark: avoid global variables>
It is tempting to declare a variable globally once and use it in all functions without redelaring it. However, this is a bad practice, because it could lead to the errors that are hard to debug, when the global variables are modified. Avoid using global variables. Using global constants is fine, since constants are never changed.

5.13.1 The Scope of Variables in a for Loop
<Side Remark: for loop control variable>
A variable declared in the initial action part of a for loop header has its scope in the entire loop. But a variable declared inside a for loop body has its scope limited in the loop body from its declaration to the end of the block that contains the variable, as shown in Figure 5.5.

```c
void function1() {
    int i;
    for (int i = 1; i < 10; i++)
        int j;
    }
```

Figure 5.5
A variable declared in the initial action part of a for loop header has its scope in the entire loop.

<Side Remark: multiple declarations>
It is commonly acceptable to declare a local variable with the same name multiple times in different non-nesting blocks in a function, as shown in Figure 5.6(a), but it is not a good practice to declare a local variable twice in nested blocks, as shown in Figure 5.6(b). In this case, i is declared in the function block and also in the for loop. The program can compile and run, but it is easy to make mistakes. So, you should avoid declaring the same variable in nested blocks.

```c
void function1()
{
    int x = 1;
    int y = 1;
    for (int i = 1; i < 10; i++)
    {
        x += i;
        y += i;
    }
}

for (int i = 1; i < 10; i++)
{
    cout << i << endl;
    cout << sum << endl;
}
```

Figure 5.6
(a) It is fine to declare i in two non-nesting blocks
(b) It is not a good practice to declare i in two nesting blocks
A variable can be declared multiple times in non-nested blocks, but you should avoid declaring them in nested blocks.

**CAUTION**

Do not declare a variable inside a block and then attempt to use it outside the block. Here is an example of a common mistake:

```cpp
for (int i = 0; i < 10; i++)
{
    cout << i;
}
```

The last statement would cause a syntax error because variable `i` is not defined outside of the `for` loop.

***End of CAUTION***

5.13.2 Static Local Variables

*Side Remark: for loop control variable*

After a function completes its execution, all its local variables are destroyed. Sometimes, it is desirable to retain the value stored in local variables so that they can be used in the next call. C++ allows you to declare static local variables. Static local variables are permanently allocated in the memory for the lifetime of the program. To declare a static variable, use the keyword `static`.

Listing 5.15 demonstrates using static local variables.

```cpp
# include <iostream>
using namespace std;

void t1(); // function prototype

int main()
{
    t1();
    t1();
    return 0;
}
```

Listing 5.15 StaticVariableDemo.cpp (Using Static Local Variables)

***PD: Please add line numbers in the following code***
```cpp
t1()
{
    static int x = 1;
    int y = 1;
    x++;
    y++;
    cout << "x is " << x << endl;
    cout << "y is " << y << endl;
}
```

```
x is 1
y is 1
x is 2
y is 1
```

A static local variable `x` is declared in line 15 with initial value 1. When `t1()` is invoked for the first time in line 8, `x` is incremented to 2 (line 17). Since `x` is a static local variable, `x` is retained in memory after this call. When `t1()` is invoked again in line 9, `x` is 2 and is incremented to 3 (line 17).

A local variable `y` is declared in line 16 with initial value 1. When `t1()` is invoked for the first time in line 8, `y` is incremented to 2 (line 18). Since `y` is a local variable, `y` is destroyed after this call. When `t1()` is invoked again in line 9, `y` is initialized to 1 and is incremented to 2 (line 18).

### 5.14 The Math Functions

C++ contains the functions needed to perform basic mathematical operations. You have already used the `pow(a, b)` function to compute `a^b` in Listing 2.3, “Computing Loan Payments.” This section introduces more useful mathematical functions, summarized in Table 5.1.

**Table 5.1**

Mathematical Functions
### Function Abstraction and Stepwise Refinement

#### <Side Remark: function abstraction>
#### <Side Remark: information hiding>

The key to developing software is to apply the concept of abstraction. You will learn many levels of abstraction from this book. Function abstraction is achieved by separating the use of a function from its implementation. The client can use a function without knowing how it is implemented. The details of the implementation are encapsulated in the function and hidden from the client who invokes the function. This is known as information hiding or encapsulation. If you decide to change the implementation, the client program will not be affected, provided that you do not change the function signature. The implementation of the function is hidden from the client in a “black box”, as shown in Figure 5.7.

### Same as Fig 4.10 in intro5e p138
Figure 5.7
The function body can be thought of as a black box that contains the detailed implementation for the function.

You have already used the `rand()` function to return a random number, the `time(0)` function to obtain the current time, and the `max` function to find the maximum number. You know how to write the code to invoke these functions in your program, but as a user of these functions, you are not required to know how they are implemented.

<Side Remark: divide and conquer>
<Side Remark: stepwise refinement>
The concept of function abstraction can be applied to the process of developing programs. When writing a large program, you can use the “divide and conquer” strategy, also known as stepwise refinement, to decompose it into subproblems. The subproblems can be further decomposed into smaller, more manageable problems.

Suppose you write a program that displays the calendar for a given month of the year. The program prompts the user to enter the year and the month, and then displays the entire calendar for the month, as shown in Figure 5.8.

Figure 5.8

After prompting the user to enter the year and the month, the program displays the calendar for that month.

Let us use this example demonstrate the divide-and-conquer approach.

5.15.1 Top-Down Design
How would you get started on such a program? Would you immediately start coding? Beginning programmers often start by trying to work out the solution to every detail. Although details are important in the final program, concern for detail in the early stages may block the problem-solving process. To make problem-solving flow as smoothly as possible, this example begins by using function abstraction to isolate details from design and only later implements the details.

For this example, the problem is first broken into two subproblems: get input from the user, and print the calendar for the month. At this stage, the creator of the program should be concerned with what the subproblems will achieve, not with how to get input and print the calendar for the month. You can draw a structure chart to help visualize the decomposition of the problem (see Figure 5.9(a)).

***Same as Fig 4.13 in intro5e p144

```
printCalendar
  (main)

  readInput

  printMonth

(a)
```

```
printMonth

  printMonthTitle

  printMonthBody

(b)
```

**Figure 5.9**

The structure chart shows that the `printCalendar` problem is divided into two subproblems, `readInput` and `printMonth`, and that `printMonth` is divided into two smaller subproblems, `printMonthTitle` and `printMonthBody`.

You can use the `cin` object to read input for the year and the month. The problem of printing the calendar for a given month can be broken into two subproblems: print the month title, and print the month body, as shown in Figure 5.10(b). The month title consists of three lines: month and year, a dash line, and the names of the seven days of the week. You need to get the month name (e.g., January) from the numeric month (e.g., 1). This is accomplished in `printMonthName` (see Figure 5.10(a)).

```
printMonthTitle

  printMonthName

(a)
```

```
printMonthBody

  getStartDay

  getNumberOfDaysInMonth

(b)
```

**Figure 5.10**
To printMonthTitle, you need printMonthName. (b) The printMonthBody problem is refined into several smaller problems.

In order to print the month body, you need to know which day of the week is the first day of the month (getStartDay) and how many days the month has (getNumberOfDaysInMonth), as shown in Figure 5.10(b). For example, December 2005 has thirty-one days, and the first of the month is Thursday, as shown in Figure 5.8.

How would you get the start day for the first date in a month? There are several ways to find the start day. Assume that you know that the start day (startDay1800 = 3) for Jan 1, 1800 was Wednesday. You could compute the total number of days (totalNumberOfDays) between Jan 1, 1800 and the first date of the calendar month. The start day for the calendar month is (totalNumberOfDays + startDay1800) % 7, since every week has seven days. So the getStartDay problem can be further refined as getTotalNumberOfDays, as shown in Figure 5.11(a).

(a) To getStartDay, you need getTotalNumberOfDays. (b) The getTotalNumberOfDays problem is refined into two smaller problems.

To get the total number of days, you need to know whether a year is a leap year and the number of days in each month. So the getTotalNumberOfDays is further refined into two subproblems: isLeapYear and getNumberOfDaysInMonth, as shown in Figure 5.11(b). The complete structure chart is shown in Figure 5.12.
Figure 5.12

The structure chart shows the hierarchical relationship of the subproblems in the program.

5.15.2 Top-Down or Bottom-Up Implementation

<side remark: top-down approach>
<side remark: stub>
<side remark: bottom-up approach>

Now we turn our attention to implementation. In general, a subproblem corresponds to a function in the implementation, although some are so simple that this is unnecessary. You would need to decide which modules to implement as functions and which to combine in other functions. Decisions of this kind should be based on whether the overall program will be easier to read as a result of your choice. In this example, the subproblem readInput can be simply implemented in the main function.

You can use either a "top-down" approach or a "bottom-up" approach. The top-down approach implements one function in the structure chart at a time from the top to the bottom. Stubs can be used for the functions waiting to be implemented. A stub is a simple but incomplete version of a function. The use of stubs enables you to test invoking the function from a caller. Implement the main function first, and then use a stub for the printMonth function. For example, let printMonth display the year and the month in the stub. Thus, your program may begin like this:

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

void printMonth(int year, int month);
void printMonthTitle(int year, int month);
```
void printMonthName(int month);
void printMonthBody(int year, int month);
int getStartDay(int year, int month);
int getTotalNumberOfDays(int year, int month);
int getNumberOfDaysInMonth(int year, int month);
bool isLeapYear(int year);

int main()
{
    // Prompt the user to enter year
    cout << "Enter full year (e.g., 2001): ";
    int year;
    cin >> year;

    // Prompt the user to enter month
    cout << "Enter month in number between 1 and 12: ";
    int month;
    cin >> month;

    // Print calendar for the month of the year
    printMonth(year, month);

    return 0;
}

void printMonth(int year, int month)
{
    cout << month << "  " << year << endl;
}

Compile and test the program, and fix any errors. You can now implement the printMonth function. For functions invoked from the printMonth function, you can again use stubs.

The bottom-up approach implements one function in the structure chart at a time from the bottom to the top. For each function implemented, write a test program to test it. The top-down and bottom-up approaches are both fine. Both approaches implement functions incrementally, help to isolate programming errors, and make debugging easy. Sometimes they can be used together.

5.15.3 Implementation Details

The isLeapYear(int year) function can be implemented using the following code:

    return (year % 400 == 0 || (year % 4 == 0 && year % 100 != 0));

Use the following fact to implement
getTotalNumberOfDaysInMonth(int year, int month):

- January, March, May, July, August, October, and December have thirty-one days.
- April, June, September, and November have thirty days.
- February has twenty-eight days during a regular year and twenty-nine days during a leap year. A regular year, therefore, has 365 days, whereas a leap year has 366 days.

To implement getTotalNumberOfDays(int year, int month), you need to compute the total number of days (totalNumberOfDays) between January 1, 1800 and the first day of the calendar month. You could find the total number of days between the year
1800 and the calendar year and then figure out the total number of days prior to the calendar month in the calendar year. The sum of these two totals is `totalNumberOfDays`.

To print a body, first pad some space before the start day and then print the lines for every week, as shown for December 2005 (see Figure 5.8).

The complete program is given in Listing 5.16.

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

// Function prototypes
void printMonth(int year, int month);
void printMonthTitle(int year, int month);
void printMonthName(int month);
void printMonthBody(int year, int month);
int getStartDay(int year, int month);
int getTotalNumberOfDays(int year, int month);
int getNumberOfDaysInMonth(int year, int month);
bool isLeapYear(int year);

int main()
{
    // Prompt the user to enter year
    cout << "Enter full year (e.g., 2001): ";
    int year;
    cin >> year;

    // Prompt the user to enter month
    cout << "Enter month in number between 1 and 12: ";
    int month;
    cin >> month;

    // Print calendar for the month of the year
    printMonth(year, month);

    return 0;
}

// Print the calendar for a month in a year
void printMonth(int year, int month)
{
    // Print the headings of the calendar
    printMonthTitle(year, month);

    // Print the body of the calendar
    printMonthBody(year, month);
}
```

Listing 5.16 PrintCalendar.cpp (Printing Calendar)
// Print the month title, e.g., May, 1999
void printMonthTitle(int year, int month)
{
  printMonthName(month);
  cout << " " << year << endl;
  cout << "-----------------------------" << endl;
  cout << " Sun Mon Tue Wed Thu Fri Sat" << endl;
}

// Get the English name for the month
void printMonthName(int month)
{
  switch (month)
  {
  case 1:
    cout << "January";
    break;
  case 2:
    cout << "February";
    break;
  case 3:
    cout << "March";
    break;
  case 4:
    cout << "April";
    break;
  case 5:
    cout << "May";
    break;
  case 6:
    cout << "June";
    break;
  case 7:
    cout << "July";
    break;
  case 8:
    cout << "August";
    break;
  case 9:
    cout << "September";
    break;
  case 10:
    cout << "October";
    break;
  case 11:
    cout << "November";
    break;
  case 12:
    cout << setw(16) << "December";
    break;
  }
}

// Print month body
void printMonthBody(int year, int month)
{
  // Get start day of the first day in the month
  int startDay = getStartDay(year, month);

  // Get number of days in the month
  int numberOfDaysInMonth = getNumberOfDaysInMonth(year, month);

  // Pad space before the first day of the month
  int i = 0;
  for (i = 0; i < startDay; i++)
    cout << " ";

  for (i = 1; i <= numberOfDaysInMonth; i++)
    cout << setw(4) << i;

  if ((i + startDay) % 7 == 0)
    cout << endl;

  // Get the start day of the first day in a month

int getStartDay(int year, int month)
    // Get total number of days since 1//1//1800
    int startDay1800 = 3;
    int totalNumberOfDays = getTotalNumberOfDays(year, month);
    // Return the start day
    return (totalNumberOfDays + startDay1800) % 7;

// Get the total number of days since January 1, 1800
int getTotalNumberOfDays(int year, int month)
    int total = 0;
    // Get the total days from 1800 to year - 1
    for (int i = 1800; i < year; i++)
        if (isLeapYear(i))
            total = total + 366;
        else
            total = total + 365;
    // Add days from Jan to the month prior to the calendar month
    for (int i = 1; i < month; i++)
        total = total + getNumberOfDaysInMonth(year, i);
    return total;

// Get the number of days in a month
int getNumberOfDaysInMonth(int year, int month)
    if (month == 1 || month == 3 || month == 5 || month == 7 ||
        month == 8 || month == 10 || month == 12)
        return 31;
    if (month == 4 || month == 6 || month == 9 || month == 11)
        return 30;
    if (month == 2) return isLeapYear(year) ? 29 : 28;
    return 0; // If month is incorrect

// Determine if it is a leap year
bool isLeapYear(int year)
    return year % 400 == 0 || (year % 4 == 0 && year % 100 != 0);

The program does not validate user input. For instance, if the user enters a month not in the range between 1 and 12, or a year before 1800, the program would display an erroneous calendar. To avoid this error, add an if statement to check the input before printing the calendar.

This program prints calendars for a month but could easily be modified to print calendars for a whole year. Although it can only print months after January 1800, it could be modified to trace the day of a month before 1800.

NOTE

Function abstraction modularizes programs in a neat, hierarchical manner. Programs written as collections of concise functions are easier to write, debug, maintain, and modify than would otherwise be the case. This writing style also promotes function reusability.
TIP

When implementing a large program, use the top-down or bottom-up approach. Do not write the entire program at once. This approach seems to take more time for coding (because you are repeatedly compiling and running the program), but it actually saves time and makes debugging easier.

Key Terms

***PD: Please place terms in two columns same as in intro5e.

• actual parameter 127
• argument 127
• ambiguous invocation 135
• divide and conquer 143
• formal parameter (i.e., parameter) 126
• information hiding 138
• function
• function abstraction 138
• function overloading 133
• function prototype 133
• function signature 126
• global variables 131
• header file 131
• local variables 131
• pass-by-reference 131
• pass-by-value 131
• parameter 131
• return type 126
• return value 126
• scope of variable 137
• static local variables 131
• stepwise refinement 143
• stub 145

Chapter Summary

• Making programs modular and reusable is one of the central goals in software engineering. C++ provides many powerful constructs that help to achieve this goal. Functions are one such construct.

• The function header specifies the return value type, function name, and parameters of the function.
• A function may return a value. The `returnValueType` is the data type of the value the function returns. If the function does not return a value, the `returnValueType` is the keyword `void`.

• The parameter list refers to the type, order, and number of the parameters of a function. The function name and the parameter list together constitute the function signature. Parameters are optional; that is, a function may contain no parameters.

• A return statement can also be used in a void function for terminating the function and returning to the function’s caller. This is useful occasionally for circumventing the normal flow of control in a function.

• The arguments that are passed to a function should have the same number, type, and order as the parameters in the function definition.

• C++ supports two ways of passing parameters: pass by value and pass by reference. Pass-by-value is to pass the value of the argument to the parameter. Pass-by-reference is to pass the reference of the argument.

• When a program calls a function, program control is transferred to the called function. A called function returns control to the caller when its return statement is executed or when its function-ending closing brace is reached.

• A function with a nonvoid return value type can also be invoked as a statement in C++. In this case, the caller simply ignores the return value. In the majority of cases, a call to a function with return value is treated as a value. In some cases, however, the caller is not interested in the return value.

• Each time a function is invoked, the system stores parameters, local variables, and system registers in a space known as a stack. When a function calls another function, the caller’s stack space is kept intact, and new space is created to handle the new function call. When a function finishes its work and returns to its caller, its associated space is released.

• A function can be overloaded. This means that two functions can have the same name as long as their function parameter lists differ.

• The scope of a variable is the part of the program where the variable can be referenced. C++ has global variables, local variables, and static local variables. Global variables are defined outside all functions and are
accessible to all functions in its scope. Local variables are defined inside a function. After a function completes its execution, all its local variables are destroyed. Static local variables can be defined to retain the local variables to be used by the next function call.

- **Function abstraction** is achieved by separating the use of a function from its implementation. The client can use a function without knowing how it is implemented. The details of the implementation are encapsulated in the function and hidden from the client who invokes the function. This is known as **information hiding** or **encapsulation**.

- Function abstraction modularizes programs in a neat, hierarchical manner. Programs written as collections of concise functions are easier to write, debug, maintain, and modify than would otherwise be the case. This writing style also promotes function reusability.

- When implementing a large program, use the top-down or bottom-up coding approach. Do not write the entire program at once. This approach seems to take more time for coding (because you are repeatedly compiling and running the program), but it actually saves time and makes debugging easier.

### Review Questions

**Sections 5.2 – 5.3**

5.1 What are the benefits of using a function? How do you declare a function? How do you invoke a function?

5.2 What is the **return** type of a **main** function?

5.3 Can you simplify the **max** function in Listing 5.1 using the conditional operator?

5.4 True of false? A call to a function with a **void** return type is always a statement itself, but a call to a function with a nonvoid return type is always a component of an expression.

5.5 What would be wrong with not writing a **return** statement in a nonvoid function? Can you have a **return** statement in a **void** function, such as the following?

```cpp
void p()  
  
  int i, 
  while (true)
```
// Prompt the user to enter an integer
cout << "Enter an integer: ";
cin >> i;

if (i == 0)
    return;

    cout << "i is " << i << endl;
}

Does the return statement in the following function cause syntax errors?

void p(double x, double y)
    i
        cout << x << " " << y << endl;
        return x + y;
}

5.6
Describe the terms parameter, argument, function signature.

5.7
Write a function header for the following functions:

• Computing a sales commission, given the sales amount and the commission rate.

• Printing the calendar for a month, given the month and year.

• Computing a square root.

• Testing whether a number is even, and returning true if it is.

• Printing a character a specified number of times.

• Computing the monthly payment, given the loan amount, number of years, and annual interest rate.

• Finding the corresponding uppercase letter, given a lowercase letter.

5.8
Identify and correct the errors in the following program:

***PD: Please add line numbers in the following code***

int xFunction(int n)
    |
    cout << n;
    |

    function1(int n, m)
    |
    n += m;
    |
    xFunction(3.4);
    |

201
Section 5.4 Passing Parameters

5.9
What is pass by value? What is pass by reference? Show the result of the following programs:

(a)
```cpp
#include <iostream>
using namespace std;

void maxValue(int value1, int value2, int max)
{
    if (value1 > value2)
        max = value1;
    else
        max = value2;
}

int main()
{
    int max = 0;
    maxValue(1, 2, max);
    cout << "max is " << max << endl;
    return 0;
}
```

(b)
```cpp
#include <iostream>
using namespace std;

void maxValue(int value1, int value2, int &max)
{
    if (value1 > value2)
        max = value1;
    else
        max = value2;
}

int main()
{
    int max = 0;
    maxValue(1, 2, max);
    cout << "max is " << max << endl;
    return 0;
}
```

(c)
```cpp
#include <iostream>
using namespace std;

void f(int i, int num)
{
    for (int j = 1; j <= i; j++)
    {
        cout << num << " ";
        num *= 2;
    }
    cout << endl;
}

int main()
{
    int i = 1;
    while (i <= 6)
    {
        f(i, 2);
        i++;
    }
    return 0;
}
```

(d)
```cpp
#include <iostream>
using namespace std;

void f(int &i, int num)
{
    for (int j = 1; j <= i; j++)
    {
        cout << num << " ";
        num *= 2;
    }
    cout << endl;
}

int main()
{
    int i = 1;
    while (i <= 6)
    {
        f(i, 2);
        i++;
    }
    return 0;
}
```

5.10
For (a) in the preceding question, show the contents of the stack just before the function `max` is invoked, just entering `max`, just before `max` is returned, and right after `max` is returned.

Section 5.5 Overloading Functions

5.11
What is function overloading? Is it possible to define two functions that have the same name but different parameter types? Is it possible to define two functions in one program.
that have identical function names and parameter lists but with different return value types?

5.12
What is wrong in the following program?

```cpp
void p(int i)
{
    cout << i << endl;
}

int p(int j)
{
    cout << j << endl;
}
```

5.13
What is wrong in the following program?

```cpp
#include <iostream>
using namespace std;

void p(int &i)
{
    cout << i << endl;
}

int p(int j)
{
    cout << j << endl;
}

int main()
{
    int k = 5;
    p(k);
    return 0;
}
```

Section 5.7 The Scope of Local Variables

5.14
Identify global variables and local variables in the following program. Does a global variable have a default value? Does a local variable have a default value? What will be the output of the following code?

```cpp
#include <iostream>
using namespace std;

int j;

int main()
{
    int i;
    cout << "i is " << i << endl;
    cout << "j is " << j << endl;
}
```
5.15
Identify global variables, local variables and static local variables in the following program. What will be the output of the following code?

```cpp
#include <iostream>
using namespace std;

int j = 40;

void p()
{
    int i = 5;
    static int j = 5;
    i++;
    j++;

    cout << "i is " << i << endl;
    cout << "j is " << j << endl;
}

int main()
{
    p();
    p();
}
```

5.16
Identify and correct the errors in the following program:

```cpp
void p(int i)
{
    int i = 5;

    cout << "i is " << i << endl;
}
```

Section 5.14 The Math Functions

5.17
True or false? The argument for trigonometric functions represents an angle in radians.

5.18
Write an expression that returns a random integer between 34 and 55. Write an expression that returns a random integer between 0 and 999. Write an expression that returns a random lowercase letter.

5.19
Assume PI is 3.14159 and E is 2.71828. Evaluate the following function calls:
A. \( \sqrt{4.0} \)  
B. \( \sin(2 \times \pi) \)  
C. \( \cos(2 \times \pi) \)  
D. \( \text{pow}(2, 2) \)  
E. \( \log(e) \)  
F. \( \exp(1.0) \)  
G. \( \max(2, \min(3, 4)) \)  
H. \( \text{fmod}(2.5, 2.3) \)  
I. \( \text{ceil}(-2.5) \)  
J. \( \text{floor}(-2.5) \)  
K. \( \text{abs}(-2.5f) \)  
L. \( \text{log10}(100.0) \)  
M. \( \cos(\pi) \)  
N. \( \text{ceil}(2.5) \)  
O. \( \text{floor}(2.5) \)  
P. \( \text{pow}(2.0, 4) \)  
Q. \( \text{fmod}(4.2, 3.5) \)  
R. \( \text{ceil}(|\text{abs}(-2.5)|) \)

**Programming Exercises**

**Sections 5.2–5.11**

5.1  
(Converting an uppercase letter to lowercase) Write a function that converts an uppercase letter to a lowercase letter. Use the following function header:

\[ \text{char upperCaseToLowerCase(char ch)} \]

For example, \( \text{upperCaseToLowerCase('B')} \) returns \( b \). See Exercise 2.7 on how to convert an uppercase letter to lowercase.

5.2*  
(Summing the digits in an integer) Write a function that computes the sum of the digits in an integer. Use the following function header:

\[ \text{int sumDigits(long n)} \]

For example, \( \text{sumDigits(234)} \) returns \( 2 + 3 + 4 = 9 \).  
HINT

Use the \% operator to extract digits, and the \(/\) operator to remove the extracted digit. For instance, to extract 4 from 234, use 234 \% 10 (=4). To remove 4 from 234, use 234 / 10 (= 23). Use a loop to repeatedly extract and remove the digit until all the digits are extracted.

5.3*  
(Displaying an integer reversed) Write the following function to display an integer in reverse order:

\[ \text{void reverse(int number)} \]

For example, \( \text{reverse(3456)} \) displays 6543.

5.4**  
(Returning an integer reversed) Write the following function to return an integer reversed:

\[ \text{int reverse(int number)} \]

For example, \( \text{reverse(3456)} \) returns 6543.

5.5*  
(Sorting three numbers) Write the following function to sort three numbers in increasing order:

\[ \text{void sort(double &num1, double &num2, double &num3)} \]
(Displaying patterns) Write the following function to display a pattern as follows:

```c
void displayPattern(int n)
```

```
   1
  2 1
 3 2 1
...
n n-1 ... 3 2 1
```

5.7*

(Computing the future investment value) Write a function that computes future investment value at a given interest rate for a specified number of years. The future investment is determined using the formula in Exercise 2.9.

Use the following function header:

```c
double futureInvestmentValue(double investmentAmount, double monthlyInterestRate, int years)
```

For example, `futureInvestmentValue(10000, 0.05/12, 5)` returns 12833.59.

Write a test program that prompts the user to enter the investment amount (e.g., 1000) and the interest rate (e.g., 9%), and print a table that displays future value for the years from 1 to 30, as shown below:

```
The amount invested: 1000
Annual interest rate: 9%

<table>
<thead>
<tr>
<th>Years</th>
<th>Future Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1093.8</td>
</tr>
<tr>
<td>2</td>
<td>1196.41</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>13467.25</td>
</tr>
<tr>
<td>30</td>
<td>14730.57</td>
</tr>
</tbody>
</table>
```

5.8

(Conversions between Celsius and Fahrenheit) Write a header file that contains the following two functions:

```c
/* Converts from Celsius to Fahrenheit */
double celsiusToFahrenheit(double celsius)

/* Converts from Fahrenheit to Celsius */
double fahrenheitToCelsius(double fahrenheit)
```

The formula for the conversion is:

```
fahrenheit = (9.0 / 5) * celsius + 32
```

Implement the header file and write a test program that invokes these functions to display the following tables:

```
<table>
<thead>
<tr>
<th>Celsius</th>
<th>Fahrenheit</th>
<th>Fahrenheit</th>
<th>Celsius</th>
</tr>
</thead>
<tbody>
<tr>
<td>40.0</td>
<td>104.0</td>
<td>120.0</td>
<td>48.89</td>
</tr>
<tr>
<td>39.0</td>
<td>102.2</td>
<td>119.0</td>
<td>48.03</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32.0</td>
<td>89.6</td>
<td>40.0</td>
<td>5.44</td>
</tr>
<tr>
<td>31.0</td>
<td>87.8</td>
<td>30.0</td>
<td>-1.11</td>
</tr>
</tbody>
</table>
```
5.9
(Conversions between feet and meters) Write a header file that contains the following two functions:

```c
/* Converts from feet to meters */
double footToMeter(double foot)

/* Converts from meters to feet */
double meterToFoot(double meter)
```

The formula for the conversion is:

```
meter = 0.305 * foot
```

Implement the header file and write a test program that invokes these functions to display the following tables:

<table>
<thead>
<tr>
<th>Feet</th>
<th>Meters</th>
<th>Feet</th>
<th>Meters</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0.305</td>
<td>20.0</td>
<td>65.574</td>
</tr>
<tr>
<td>2.0</td>
<td>0.61</td>
<td>25.0</td>
<td>81.967</td>
</tr>
<tr>
<td>...</td>
<td>2.745</td>
<td>60.0</td>
<td>195.721</td>
</tr>
<tr>
<td>10.0</td>
<td>3.05</td>
<td>65.0</td>
<td>213.115</td>
</tr>
</tbody>
</table>

5.10
(Computing GCD) Write a function that returns the greatest common divisor between two positive integers using the following header:

```
int gcd(int m, int n)
```

Write a test program that computes \( \text{gcd}(24, 16) \) and \( \text{gcd}(255, 25) \).

5.11
(Computing commissions) Write a function that computes the commission using the scheme in Listing 3.11, “Finding Sales Amount.” The header of the function is as follows:

```
double computeCommission(double salesAmount)
```

Write a test program that displays the following table:

<table>
<thead>
<tr>
<th>SalesAmount</th>
<th>Commission</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>900.0</td>
</tr>
<tr>
<td>15000</td>
<td>1500.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>95000</td>
<td>11100.0</td>
</tr>
<tr>
<td>100000</td>
<td>11700.0</td>
</tr>
</tbody>
</table>

5.12
(Displaying characters) Write a function that prints characters using the following header:

```
void printChars(char ch1, char ch2, int numberPerLine)
```

This function prints the characters between \( \text{ch1} \) and \( \text{ch2} \) with the specified numbers per line. Write a test program that prints ten characters per line from '1' and 'Z'.

207
5.13* (Summing series) Write a function to compute the following series:

\[ m(i) = \frac{1}{2} + \frac{2}{3} + \cdots + \frac{i}{i+1} \]

Write a test program that displays the following table:

<table>
<thead>
<tr>
<th>i</th>
<th>m(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>1.1667</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>15.4523</td>
</tr>
<tr>
<td>20</td>
<td>16.4023</td>
</tr>
</tbody>
</table>

5.14* (Computing series) Write a function to compute the following series:

\[ m(i) = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots - \frac{1}{2i-1} + \frac{1}{2i+1}\right) \]

5.15* (Printing a tax table) Use the computeTax functions in Listing 5.4, “Computing Tax,” to write a program that prints a 2002 tax table for taxable income from $50,000 to $60,000 with intervals of $50 for all four statuses, as follows:

<table>
<thead>
<tr>
<th>taxable Income</th>
<th>Single</th>
<th>Married</th>
<th>Married</th>
<th>Head of a House</th>
</tr>
</thead>
<tbody>
<tr>
<td>50000</td>
<td>9846</td>
<td>7296</td>
<td>10398</td>
<td>8506</td>
</tr>
<tr>
<td>50050</td>
<td>9859</td>
<td>7309</td>
<td>10411</td>
<td>8519</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>59950</td>
<td>12532</td>
<td>9982</td>
<td>13190</td>
<td>11192</td>
</tr>
<tr>
<td>60000</td>
<td>12546</td>
<td>9996</td>
<td>13205</td>
<td>11206</td>
</tr>
</tbody>
</table>

5.16* (Revising Example 3.13 “Printing Prime Numbers”) Write a program that meets the following requirements:

[BL] Declare a function to determine whether an integer is a prime number. Use the following function header:

```java
boolean isPrime(int num)
```

An integer greater than 1 is a prime number if its only divisor is 1 or itself. For example, `isPrime(11)` returns true, and `isPrime(9)` returns false.

[BL] Use the `isPrime` function to find the first thousand prime numbers and display every ten prime numbers in a row, as follows:

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>11</th>
<th>13</th>
<th>17</th>
<th>19</th>
<th>23</th>
<th>29</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>37</td>
<td>41</td>
<td>43</td>
<td>47</td>
<td>53</td>
<td>59</td>
<td>61</td>
<td>67</td>
<td>71</td>
</tr>
<tr>
<td>73</td>
<td>79</td>
<td>83</td>
<td>89</td>
<td>97</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.17*
(Displaying matrix of 0s and 1s) Write a function that displays an n by n matrix using the following header:

```c
void printMatrix(int n);
```

Each element is 0 or 1, which is generated randomly. Write a test program that prints a 3 by 3 matrix that may look like this:

```
0 1 0
0 0 0
1 1 1
```

**Section 5.12**

5.18*

(Generating random characters) Use the functions in RandomCharacter in Listing 5.12 to print one hundred uppercase letters and then one hundred single digits, and print ten per line.

**Section 5.14 The Math Functions**

5.19

(Using the sqrt function) Write a program that prints the following table using the sqrt function.

```
<table>
<thead>
<tr>
<th>Number</th>
<th>SquareRoot</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0000</td>
</tr>
<tr>
<td>2</td>
<td>1.4142</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>5.2426</td>
</tr>
<tr>
<td>20</td>
<td>5.4721</td>
</tr>
</tbody>
</table>
```

5.20*

(The MyTriangle header file) Create a header file named MyTriangle.h that contains the following two functions:

```c
/* Returns true if the sum of any two sides is greater than the third side. */
boolean isValid(double side1, double side2, double side3);

/* Returns the area of the triangle. */
double area(double side1, double side2, double side3);
```

The formula for computing the area is

\[
s = \frac{(side1 + side2 + side3)}{2};
\]

\[
area = \sqrt{s(s-side1)(s-side2)(s-side3)}
\]

Implement the header file and write a test program that reads three sides for a triangle and computes the area if the input is valid. Otherwise, display that the input is invalid.

5.21
(Using trigonometric functions) Print the following table to display the \( \sin \) value and \( \cos \) value of degrees from 0 to 360 with increments of 10 degrees. Round the value to keep four digits after the decimal point.

<table>
<thead>
<tr>
<th>Degree</th>
<th>( \sin )</th>
<th>( \cos )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>10</td>
<td>0.1736</td>
<td>0.9848</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>350</td>
<td>-0.1736</td>
<td>0.9848</td>
</tr>
<tr>
<td>360</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

5.22**

(Computing mean and standard deviation) In business applications, you are often asked to compute the mean and standard deviation of data. The mean is simply the average of the numbers. The standard deviation is a statistic that tells you how tightly all the various data are clustered around the mean in a set of data. For example, what is the average age of the students in a class? How close are the ages? If all the students are the same age, the deviation is 0. Write a program that generates ten random numbers between 0 and 1000, and computes the mean and standard deviations of these numbers using the following formula:

\[
\text{mean} = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{x_1 + x_2 + ... + x_n}{n}
\]

\[
\text{deviation} = \sqrt{\frac{\sum_{i=1}^{n} x_i^2 - \left(\frac{\sum_{i=1}^{n} x_i}{n}\right)^2}{n - 1}}
\]

5.23**

(Approximating the square root) Implement the \( \text{sqrt} \) function. The square root of a number, \( \text{num} \), can be approximated by repeatedly performing a calculation using the following formula:

\[
\text{nextGuess} = \frac{(\text{lastGuess} + (\text{num} / \text{lastGuess}))}{2}
\]

When \( \text{nextGuess} \) and \( \text{lastGuess} \) are almost identical, \( \text{nextGuess} \) is the approximated square root.

The initial guess will be the starting value of \( \text{lastGuess} \). If the difference between \( \text{nextGuess} \) and \( \text{lastGuess} \) is less than a very small number, such as 0.0001, you can claim that \( \text{nextGuess} \) is the approximated square root of \( \text{num} \).

5.24**

(Displaying current date and time) Listing 2.5, “Displaying Current Time,” displays the current time. Improve this example to display the current date and time. The calendar example in §5.15, “Function Abstraction and Stepwise Refinement,” should give you some ideas on how to find year, month, and day.