Chapter Five
Modular Programming

Modular programming and Modules

Modular programming is breaking down the design of a program into individual components (modules) that can be programmed and tested independently. It is a requirement for effective development and maintenance of large programs and projects. With modular programming, procedures of a common functionality are grouped together into separate *modules*. A program therefore no longer consists of only one single part. It is now divided into several smaller parts which interact and which form the whole program.

5.1. Definition of Functions

Modules in C++ are called functions. A function is a subprogram that can act on data and return a value. Every C++ program has at least one function, `main()`. When your program starts, `main()` is called automatically. `main()` might call other functions, some of which might call still others. Each function has its own name, and when that name is encountered, the execution of the program branches to the body of that function. When the function returns, execution resumes on the next line of the calling function. When a program calls a function, execution switches to the function and then resumes at the line after the function call. Well-designed functions perform a specific and easily understood task. Complicated tasks should be broken down into multiple functions, and then each can be called in turn. Functions come in two varieties: user-defined and built-in. Built-in functions are part of your compiler package--they are supplied by the manufacturer for your use. In this chapter we will discuss about user-defined functions.

5.1.1. Declaring and Defining Functions

Using functions in your program requires that you first declare the function and that you then define the function. The declaration tells the compiler the name, return type, and parameters of the function. The definition tells the compiler how the function works. No function can be called from any other function that hasn't first been declared. The declaration of a function is called its prototype.

5.1.1.1. Declaring the Function

There are three ways to declare a function:

- Write your prototype into a file, and then use the `#include` directive to include it in your program.
- Write the prototype into the file in which your function is used.
- Define the function before it is called by any other function. When you do this, the definition acts as its own declaration.

Although you can define the function before using it, and thus avoid the necessity of creating a function prototype, this is not good programming practice for three reasons. First, it is a bad idea to require that functions appear in a file in a particular order. Doing so makes it hard to maintain the program as requirements change. Second, it is possible that function `A()` needs to be able to call function `B()`, but function `B()` also needs to be able to

call function `A()` under some circumstances. It is not possible to define function `A()` before you define function `B()` and also to define function `B()` before you define function `A()`, so at least one of them must be declared in any case. Third, function prototypes are a good and powerful debugging technique. If your prototype declares that your function takes a particular set of parameters, or that it returns a particular type of value, and then your function does not match the prototype, the compiler can flag your error instead of waiting for it to show itself when you run the program.

Function Prototypes

Many of the built-in functions you use have their function prototypes already written in the files you include in your program by using `#include`. For functions you write yourself, you must include the prototype. The function prototype is a statement, which means it ends with a semicolon. It consists of the function's return type, name, and parameter list. The parameter list is a list of all the parameters and their types, separated by commas. *Function Prototype Syntax:*

return_type function_name ([type [parameterName1] type [parameterName2]]...);

The function prototype and the function definition must agree exactly about the return type, the name, and the parameter list. If they do not agree, you will get a compile-time error. Note, however, that the function prototype does not need to contain the names of the parameters, just their types. A prototype that looks like this is perfectly legal:

long Area(int, int);

This prototype declares a function named `Area()` that returns a `long` and that has two parameters, both integers. Although this is legal, it is not a good idea. Adding parameter names makes your prototype clearer. The same function with named parameters might be

long Area(int length, int width);

It is now obvious what this function does and what the parameters are. Note that all functions have a return type. If none is explicitly stated, the return type defaults to `int`. Your programs will be easier to understand, however, if you explicitly declare the return type of every function, including `main()`.

Function Prototype Examples

```
long FindArea(long length, long width);      // returns long, has two parameters
void PrintMessage(int messageNumber);        // returns void, has one parameter
int GetChoice();                             // returns int, has no parameters
BadFunction();                               // returns int, has no parameters
```

Listing 5.1 demonstrates a program that includes a function prototype for the `Area()` function.

Listing 5.1. A function declaration and the definition and use of that function.

```
1:  // Listing 5.1 - demonstrates the use of function prototypes
2:
3:  typedef unsigned short USHORT;
4:  #include <iostream.h>
5:  USHORT FindArea(USHORT length, USHORT width); //function prototype
6:
7:  int main()
8:  {
9:    USHORT lengthOfYard;
10:   USHORT widthOfYard;
11:   USHORT areaOfYard;
12:
13:   cout << "\nHow wide is your yard? ";
14:   cin >> widthOfYard;
15:   cout << "\nHow long is your yard? ";
16:   cin >> lengthOfYard;
17:
18:   areaOfYard= FindArea(lengthOfYard,widthOfYard);
19:
20:   cout << "\nYour yard is ";
21:   cout << areaOfYard;
22:   cout << " square feet\n\n";
23:          return 0;
24: }
25:
26: USHORT FindArea(USHORT l, USHORT w)
27: {
28:     return l * w;
29: }
```

Output: How wide is your yard? 100

How long is your yard? 200

Your yard is 20000 square feet

Analysis: The prototype for the `FindArea()` function is on line 5. Compare the prototype with the definition of the function on line 26. Note that the name, the return type, and the parameter types are the same. If they were different, a compiler error would have been generated. In fact, the only required difference is that the function prototype ends with a semicolon and has no body. Also note that the parameter names in the prototype are `length` and `width`, but the parameter names in the definition are `l` and `w`. The names in the prototype are not used; they are there as information to the programmer. When they are included, they should match the implementation when possible. This is a matter of good programming style and reduces confusion, but it is not required, as you see here.

The arguments are passed in to the function in the order in which they are declared and defined, but there is no matching of the names. Had you passed in `widthOfYard`, followed by `lengthOfYard`, the `FindArea()` function would have used the value in `widthOfYard` for `length` and `lengthOfYard` for `width`. The body of the function is always enclosed in braces, even when it consists of only one statement, as in this case.

5.1.1.2. Defining the Function

The definition of a function consists of the function header and its body. The header is exactly like the function prototype, except that the parameters must be named, and there is no terminating semicolon. The body of the function is a set of statements enclosed in braces. *Function Definition Syntax*

```
return_type function_name ( [type parameterName1], [type parameterName2]...)
{
  statements;
}
```

A function prototype tells the compiler the return type, name, and parameter list. Functions are not required to have parameters, and if they do, the prototype is not required to list their names, only their types. A prototype always ends with a semicolon (;). A function definition must agree in return type and parameter list with its prototype. It must provide names for all the parameters, and the body of the function definition must be surrounded by braces. All statements within the body of the function must be terminated with semicolons, but the function itself is not ended with a semicolon; it ends with a closing brace. If the function returns a value, it should end with a `return` statement, although `return` statements can legally appear anywhere in the body of the function. Every function has a return type. If one is not explicitly designated, the return type will be `int`. Be sure to give every function an explicit return type. If a function does not return a value, its return type will be `void`.

Function Definition Examples

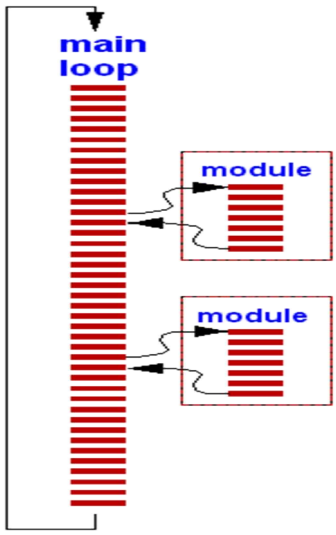
```
long Area(long l, long w)
{
     return l * w;
}

void PrintMessage(int whichMsg)
{
     if (whichMsg == 0)
          cout << "Hello.\n";
     if (whichMsg == 1)
          cout << "Goodbye.\n";
     if (whichMsg > 1)
          cout << "I'm confused.\n";
}
```

Function Statements

There is virtually no limit to the number or types of statements that can be in a function body. Although you can't define another function from within a function, you can call a function, and of course `main()` does just that in nearly every C++ program. Functions can even call themselves, which is discussed soon, in the section on recursion. Although there is no limit to the size of a function in C++, well-designed functions tend to be small. Many programmers advise keeping your functions short enough to fit on a single screen so that you can see the entire function at one time. This is a rule of thumb, often broken by very good programmers, but a smaller function is easier to understand and maintain. Each function should carry out a single, easily understood task. If your functions start getting large, look for places where you can divide them into component tasks.

Execution of Functions

When you call a function, execution begins with the first statement after the opening brace (`{`). Branching can be accomplished by using the `if` statement. Functions can also call other functions and can even call themselves (see the section "Recursion," later in this chapter). Each function has its own name, and when that name is encountered, the execution of the program branches to the body of that function. When the function returns, execution resumes on the next line of the calling function. This flow is illustrated in the following figure.



5.2. Scope of variables

A variable has scope, which determines how long it is available to your program and where it can be accessed. There are two scopes *Local* and *Global.*

5.2.1. Local Variables

Not only can you pass in variables to the function, but you also can declare variables within the body of the function. This is done using local variables, so named because they exist

only locally within the function itself. When the function returns, the local variables are no longer available. Local variables are defined like any other variables. The parameters passed in to the function are also considered local variables and can be used exactly as if they had been defined within the body of the function. Variables declared within the function are said to have "local scope." That means that they are visible and usable only within the function in which they are defined. In fact, in C++ you can define variables anywhere within the function, not just at its top. The scope of the variable is the block in which it is defined. Thus, if you define a variable inside a set of braces within the function, that variable is available only within that block. Listing 5.2 is an example of using parameters and locally defined variables within a function.

Listing 5.2. The use of local variables and parameters.

```
1:     #include <iostream.h>
2:
3:     float Convert(float);
4:     int main()
5:     {
6:        float TempFer;
7:        float TempCel;
8:
9:        cout << "Please enter the temperature in Fahrenheit: ";
10:       cin >> TempFer;
11:       TempCel = Convert(TempFer);
12:       cout << "\nHere's the temperature in Celsius: ";
13:       cout << TempCel << endl;
14:               return 0;
15:    }
16:
17:    float Convert(float TFer)
18:    {
19:       float TCel;
20:       TCel = ((TFer - 32) * 5) / 9;
21:       return TCel;
22: }
```

```
Output: Please enter the temperature in Fahrenheit: 212

Here's the temperature in Celsius: 100

Please enter the temperature in Fahrenheit: 32

Here's the temperature in Celsius: 0

Please enter the temperature in Fahrenheit: 85

Here's the temperature in Celsius: 29.4444
```

Analysis: On lines 6 and 7, two `float` variables are declared, one to hold the temperature in Fahrenheit and one to hold the temperature in degrees Celsius. The user is prompted to enter a Fahrenheit temperature on line 9, and that value is passed to the function `Convert()`.

Execution jumps to the first line of the function `Convert()` on line 19, where a local variable, named `TCel`, is declared. Note that this is local variable that exists only within the function `Convert()`. The value passed as a parameter, `TFer`, is also just a local copy of the variable passed in by `main()`.The local function variable `TCel` is assigned the value that results from subtracting 32 from the parameter `TFer`, multiplying by 5, and then dividing by 9. This value is then returned as the return value of the function, and on line 11 it is assigned to the variable `TempCel` in the `main()` function. The value is printed on line 13. The program is run three times. The first time, the value `212` is passed in to ensure that the boiling point of water in degrees Fahrenheit (212) generates the correct answer in degrees Celsius (100). The second test is the freezing point of water. The third test is a random number chosen to generate a fractional result.

Variables declared within a block are scoped to that block; they can be accessed only within that block and "go out of existence" when that block ends. Global variables have global scope and are available anywhere within your program. Normally scope is obvious, but there are some tricky exceptions. Currently, variables declared within the header of a `for` loop `(for int i = 0; i<SomeValue; i++)` are scoped to the block in which the `for` loop is created.

5.2.2. Global Variables

Variables defined outside of any function have global scope and thus are available from any function in the program, including `main()`. Local variables with the same name as global variables do not change the global variables. A local variable with the same name as a global variable hides the global variable, however. If a function has a variable with the same name as a global variable, the name refers to the local variable--not the global--when used within the function. Listing 5.3 illustrates these points.

Listing 5.3. Demonstrating global and local variables.

```
1:   #include <iostream.h>
2:   void myFunction();           // prototype
3:
4:   int x = 5, y = 7;            // global variables
5:   int main()
6:   {
7:
8:        cout << "x from main: " << x << "\n";
9:        cout << "y from main: " << y << "\n\n";
10:       myFunction();
11:       cout << "Back from myFunction!\n\n";
12:       cout << "x from main: " << x << "\n";
13:       cout << "y from main: " << y << "\n";
14:       return 0;
15:  }
16:
17:  void myFunction()
18:  {
19:       int y = 10;
20:
```

```
21:       cout << "x from myFunction: " << x << "\n";
22:       cout << "y from myFunction: " << y << "\n\n";
23: }

 Output: x from main: 5
 y from main: 7

 x from myFunction: 5
 y from myFunction: 10

 Back from myFunction!

 x from main: 5
 y from main: 7
```

Analysis: This simple program illustrates a few key, and potentially confusing, points about local and global variables. On line 1, two global variables, `x` and `y`, are declared. The global variable `x` is initialized with the value `5`, and the global variable `y` is initialized with the value `7`. On lines 8 and 9 in the function `main()`, these values are printed to the screen. Note that the function `main()` defines neither variable; because they are global, they are already available to `main()`.

When `myFunction()` is called on line 10, program execution passes to line 18, and a local variable, `y`, is defined and initialized with the value `10`. On line 21, `myFunction()` prints the value of the variable `x`, and the global variable `x` is used, just as it was in `main()`. On line 22, however, when the variable name `y` is used, the local variable `y` is used, hiding the global variable with the same name.The function call ends, and control returns to `main()`, which again prints the values in the global variables. Note that the global variable `y` was totally unaffected by the value assigned to `myFunction()`'s local `y` variable.

Listing 5.4. Variables scoped within a block.

```
1:    // Listing 5.4 - demonstrates variables
2:    // scoped within a block
3:
4:    #include <iostream.h>
5:
6:    void myFunc();
7:
8:    int main()
9:    {
10:      int x = 5;
11:      cout << "\nIn main x is: " << x;
12:
13:      myFunc();
14:
15:      cout << "\nBack in main, x is: " << x;
16:        return 0;
17:   }
18:
19:   void myFunc()
20:   {
```

```
21:
22:       int x = 8;
23:       cout << "\nIn myFunc, local x: " << x << endl;
24:
25:       {
26:          cout << "\nIn block in myFunc, x is: " << x;
27:
28:          int x = 9;
29:
30:          cout << "\nVery local x: " << x;
31:       }
32:
33:       cout << "\nOut of block, in myFunc, x: " << x << endl;
34: }
```

```
Output: In main x is: 5
In myFunc, local x: 8

In block in myFunc, x is: 8
Very local x: 9
Out of block, in myFunc, x: 8

Back in main, x is: 5
```

Analysis: This program begins with the initialization of a local variable, `x`, on line 10, in `main()`. The printout on line 11 verifies that `x` was initialized with the value `5`. `MyFunc()` is called, and a local variable, also named `x`, is initialized with the value `8` on line 22. Its value is printed on line 23. A block is started on line 25, and the variable `x` from the function is printed again on line 26. A new variable also named `x`, but local to the block, is created on line 28 and initialized with the value `9`. The value of the newest variable `x` is printed on line 30. The local block ends on line 31, and the variable created on line 28 goes "out of scope" and is no longer visible.

When `x` is printed on line 33, it is the `x` that was declared on line 22. This `x` was unaffected by the `x` that was defined on line 28; its value is still `8`. On line 34, `MyFunc()` goes out of scope, and its local variable `x` becomes unavailable. Execution returns to line 15, and the value of the local variable `x`, which was created on line 10, is printed. It was unaffected by either of the variables defined in `MyFunc()`. Needless to say, this program would be far less confusing if these three variables were given unique names!

Global Variables: A Word of Caution

In C++, global variables are legal, but they are almost never used. Globals are dangerous because they are shared data, and one function can change a global variable in a way that is invisible to another function. This can and does create bugs that are very difficult to find.

Scope resolution operator

When a local variable has the same name as a global variable, all references to the variable name made with in the scope of the local variable. This situation is illustrated in the following program.

```
# include<iostream.h>
float num = 42.8;    //global variable
int main()
{
float num = 26.4;    //local variable
cout<<"the value of num is:"<<num<<endl;
return 0;
}
```

The output of the above program is:

```
the value of num is: 26.4
```

As shown by the above output, the local variable name takes precedence ovger the global variable. In such cases, we can still access the global variable by using C++'s scope resolution operator (::). This operator must be placed immediately before the variable name, as in ::num. When used in this manner, the :: tells the compiler to use global variable.E.g

```
float num = 42.8;    //global variable
int main()
{
float num = 26.4;    //local variable
cout<<"the value of num is:"<< ::num<<endl;
return 0;
}
```

The out is:

```
   the value of the number is 42.8
```

As indicated the above output, the scope resolution operator causes the global, rather the local variable to be accessed.

5.3. Function Arguments

Function arguments do not have to all be of the same type. It is perfectly reasonable to write a function that takes an integer, two `long`s, and a character as its arguments. Any valid C++ expression can be a function argument, including constants, mathematical and logical expressions, and other functions that return a value.

Using Functions as Parameters to Functions

Although it is legal for one function to take as a parameter a second function that returns a value, it can make for code that is hard to read and hard to debug. As an example, say you have the functions `double()`, `triple()`, `square()`, and `cube()`, each of which returns a value. You could write

```
Answer = (double(triple(square(cube(myValue)))));
```

This statement takes a variable, `myValue`, and passes it as an argument to the function `cube()`, whose return value is passed as an argument to the function `square()`, whose return value is in turn passed to `triple()`, and that return value is passed to `double()`. The return value of this doubled, tripled, squared, and cubed number is now passed to `Answer`.

It is difficult to be certain what this code does (was the value tripled before or after it was squared?), and if the answer is wrong it will be hard to figure out which function failed. An alternative is to assign each step to its own intermediate variable:

```
unsigned long myValue = 2;
unsigned long cubed  =   cube(myValue);        // cubed = 8
unsigned long squared = square(cubed);         // squared = 64
unsigned long tripled = triple(squared);       // tripled = 196
unsigned long Answer =  double(tripled);       // Answer = 392
```

Now each intermediate result can be examined, and the order of execution is explicit.

5.4. Passing arguments

5.4.1. Pass by Value

The arguments passed in to the function are local to the function. Changes made to the arguments do not affect the values in the calling function. This is known as passing by value, which means a local copy of each argument is made in the function. These local copies are treated just like any other local variables. Listing 5.5 illustrates this point.

Listing 5.5. A demonstration of passing by value.

```
1:     // Listing 5.5 - demonstrates passing by value
2:
3:      #include <iostream.h>
4:
5:      void swap(int x, int y);
6:
7:      int main()
8:      {
9:        int x = 5, y = 10;
10:
11:     cout << "Main. Before swap, x: " << x << " y: " <<y<< "\n";
12:        swap(x,y);
13:     cout << "Main. After swap, x: " << x << " y: " <<y << "\n";
14:           return 0;
15:      }
16:
17:      void swap (int x, int y)
18:      {
19:        int temp;
20:
21:     cout << "Swap. Before swap, x: " << x << " y: " <<y<< "\n";
22:
23:        temp = x;
```

```
24:         x = y;
25:         y = temp;
26:
27:     cout << "Swap. After swap, x: " << x << " y: " << y << "\n";
28:
29: }
```

```
Output: Main. Before swap, x: 5 y: 10
Swap. Before swap, x: 5 y: 10
Swap. After swap, x: 10 y: 5
Main. After swap, x: 5 y: 10
```

Analysis: This program initializes two variables in `main()` and then passes them to the `swap()` function, which appears to swap them. When they are examined again in `main()`, however, they are unchanged! The variables are initialized on line 9, and their values are displayed on line 11. `swap()` is called, and the variables are passed in. Execution of the program switches to the `swap()` function, where on line 21 the values are printed again. They are in the same order as they were in `main()`, as expected. On lines 23 to 25 the values are swapped, and this action is confirmed by the printout on line 27. Indeed, while in the `swap()` function, the values are swapped. Execution then returns to line 13, back in `main()`, where the values are no longer swapped.

As you've figured out, the values passed in to the `swap()` function are passed by value, meaning that copies of the values are made that are local to `swap()`. These local variables are swapped in lines 23 to 25, but the variables back in `main()` are unaffected.

5.4.2. Pass by reference

In C++, passing by reference is accomplished in two ways: using pointers and using references. The syntax is different, but the net effect is the same. Rather than a copy being created within the scope of the function, the actual original object is passed into the function. Passing an object by reference allows the function to change the object being referred to. In previous section showed that a call to the `swap()` function did not affect the values in the calling function.

Listing 9.7. swap() rewritten with references.

```
1:     //Listing 9.7 Demonstrates passing by reference
2:      // using references!
3:
4:        #include <iostream.h>
5:
6:        void swap(int &x, int &y);
7:
8:        int main()
9:        {
10:            int x = 5, y = 10;
11:
12:     cout << "Main. Before swap, x: " << x << " y: " <<y<< "\n";
13:             swap(x,y);
```

```
14:     cout << "Main. After swap, x: " << x << " y: " <<y << "\n";
15:     return 0;
16:           }
17:
18:           void swap (int &rx, int &ry)
19:           {
20:             int temp;
21:
22:    cout << "Swap. Before swap, rx: " <<rx<< " ry: " <<ry<<"\n";
23:
24:                temp = rx;
25:                rx = ry;
26:                ry = temp;
27:
28:    cout << "Swap. After swap, rx: " <<rx<< " ry: " <<ry<< "\n";
29:
30: }
 Output: Main. Before swap, x:5 y: 10
 Swap. Before swap, rx:5 ry:10
 Swap. After swap, rx:10 ry:5
 Main. After swap, x:10, y:5
```

Anaylsis: Two variables are declared on line 10 and their values are printed on line 12. On line 13, the function `swap()` is called, but note that `x` and `y`, not their addresses, are passed. The calling function simply passes the variables. When `swap()` is called, program execution jumps to line 18, where the variables are identified as references. Their values are printed on line 22, but note that no special operators are required. These are aliases for the original values, and can be used as such. On lines 24-26, the values are swapped, and then they're printed on line 28. Program execution jumps back to the calling function, and on line 14, the values are printed in `main()`. Because the parameters to `swap()` are declared to be references, the values from `main()` are passed by reference, and thus are changed in `main()` as well. References provide the convenience and ease of use of normal variables.

5.5. Return Values

Functions return a value or return `void`. `Void` is a signal to the compiler that no value will be returned. To return a value from a function, write the keyword `return` followed by the value you want to return. The value might itself be an expression that returns a value. For example:

```
return 5;
return (x > 5);
return (MyFunction());
```

These are all legal `return` statements, assuming that the function `MyFunction()` itself returns a value. The value in the second statement, `return (x > 5)`, will be zero if `x` is not greater than 5, or it will be `1`. What is returned is the value of the expression, `0` (`false`) or `1` (`true`), not the value of `x`.

When the `return` keyword is encountered, the expression following `return` is returned as the value of the function. Program execution returns immediately to the calling function, and any statements following the return are not executed. It is legal to have more than one `return` statement in a single function. Listing 5.6 illustrates this idea.

Listing 5.6. A demonstration of multiple return statements.

```
1:     // Listing 5.6 - demonstrates multiple return
2:     // statements
3:
4:     #include <iostream.h>
5:
6:     int Doubler(int AmountToDouble);
7:
8:     int main()
9:     {
10:
11:         int result = 0;
12:         int input;
13:
14:    cout << "Enter a number between 0 and 10,000 to double: ";
15:         cin >> input;
16:
17:         cout << "\nBefore doubler is called... ";
18:   cout<< "\ninput: " << input << " doubled: " <<result<<"\n";
19:
20:         result = Doubler(input);
21:
22:         cout << "\nBack from Doubler...\n";
23:  cout<< "\ninput: " << input << "  doubled: " <<result<<"\n";
24:
25:
26:         return 0;
27:    }
28:
29:    int Doubler(int original)
30:    {
31:         if (original <= 10000)
32:              return original * 2;
33:         else
34:              return -1;
35:         cout << "You can't get here!\n";
36: }
```

```
Output: Enter a number between 0 and 10,000 to double: 9000

Before doubler is called...
input: 9000 doubled: 0

Back from doubler...

input: 9000   doubled: 18000

Enter a number between 0 and 10,000 to double: 11000

Before doubler is called...
```

```
input: 11000 doubled: 0

Back from doubler...
input: 11000  doubled: -1
```

Analysis: A number is requested on lines 14 and 15, and printed on line 18, along with the local variable result. The function `Doubler()` is called on line 20, and the input value is passed as a parameter. The result will be assigned to the local variable `result`, and the values will be reprinted on lines 22 and 23. On line 31, in the function `Doubler()`, the parameter is tested to see whether it is greater than 10,000. If it is not, the function returns twice the original number. If it is greater than 10,000, the function returns `-1` as an error value. The statement on line 35 is never reached, because whether or not the value is greater than 10,000, the function returns before it gets to line 35, on either line 32 or line 34. A good compiler will warn that this statement cannot be executed, and a good programmer will take it out!

5.6. Default Parameters

For every parameter you declare in a function prototype and definition, the calling function must pass in a value. The value passed in must be of the declared type. Thus, if you have a function declared as

```
long myFunction(int);
```

the function must in fact take an integer variable. If the function definition differs, or if you fail to pass in an integer, you will get a compiler error. The one exception to this rule is if the function prototype declares a default value for the parameter. A default value is a value to use if none is supplied. The preceding declaration could be rewritten as

```
long myFunction (int x = 50);
```

This prototype says, "`myFunction()` returns a `long` and takes an integer parameter. If an argument is not supplied, use the default value of `50`." Because parameter names are not required in function prototypes, this declaration could have been written as

```
long myFunction (int = 50);
```

The function definition is not changed by declaring a default parameter. The function definition header for this function would be

```
long myFunction (int x)
```

If the calling function did not include a parameter, the compiler would fill `x` with the default value of `50`. The name of the default parameter in the prototype need not be the same as the name in the function header; the default value is assigned by position, not name. Any or all of the function's parameters can be assigned default values. The one

restriction is this: If any of the parameters does not have a default value, no previous parameter may have a default value.

If the function prototype looks like

```
long myFunction (int Param1, int Param2, int Param3);
```

you can assign a default value to `Param2` only if you have assigned a default value to `Param3`. You can assign a default value to `Param1` only if you've assigned default values to both `Param2` and `Param3`. Listing 5.7 demonstrates the use of default values.

Listing 5.7. A demonstration of default parameter values.

```
1:   // Listing 5.7 - demonstrates use
2:   // of default parameter values
3:
4:   #include <iostream.h>
5:
6:   int AreaCube(int length, int width = 25, int height = 1);
7:
8:   int main()
9:   {
10:       int length = 100;
11:       int width = 50;
12:       int height = 2;
13:       int area;
14:
15:       area = AreaCube(length, width, height);
16:       cout << "First area equals: " << area << "\n";
17:
18:       area = AreaCube(length, width);
19:       cout << "Second time area equals: " << area << "\n";
20:
21:       area = AreaCube(length);
22:       cout << "Third time area equals: " << area << "\n";
23:        return 0;
24:   }
25:
26:   AreaCube(int length, int width, int height)
27:   {
28:
29:       return (length * width * height);
30: }
```

```
Output: First area equals: 10000
Second time area equals: 5000
Third time area equals: 2500
```

Analysis: On line 6, the `AreaCube()` prototype specifies that the `AreaCube()` function takes three integer parameters. The last two have default values. This function computes the area of the cube whose dimensions are passed in. If no `width` is passed in, a `width` of 25 is used and a `height` of 1 is used. If the `width` but not the `height` is passed in, a `height`

of 1 is used. It is not possible to pass in the `height` without passing in a `width`. On lines 10-12, the dimensions `length`, `height`, and `width` are initialized, and they are passed to the `AreaCube()` function on line 15. The values are computed, and the result is printed on line 16. Execution returns to line 18, where `AreaCube()` is called again, but with no value for `height`. The default value is used, and again the dimensions are computed and printed. Execution returns to line 21, and this time neither the `width` nor the `height` is passed in. Execution branches for a third time to line 27. The default values are used. The area is computed and then printed.

DO remember that function parameters act as local variables within the function. **DON'T** try to create a default value for a first parameter if there is no default value for the second. **DON'T** forget that arguments passed by value can not affect the variables in the calling function. **DON'T** forget that changes to a global variable in one function change that variable for all functions.

5.7. Inline Functions

When you define a function, normally the compiler creates just one set of instructions in memory. When you call the function, execution of the program jumps to those instructions, and when the function returns, execution jumps back to the next line in the calling function. If you call the function 10 times, your program jumps to the same set of instructions each time. This means there is only one copy of the function, not 10.

There is some performance overhead in jumping in and out of functions. It turns out that some functions are very small, just a line or two of code, and some efficiency can be gained if the program can avoid making these jumps just to execute one or two instructions. When programmers speak of efficiency, they usually mean speed: the program runs faster if the function call can be avoided.

If a function is declared with the keyword `inline`, the compiler does not create a real function: it copies the code from the inline function directly into the calling function. No jump is made; it is just as if you had written the statements of the function right into the calling function.

Syntax

```
inline return_type function_name ( [type parameterName1], [type
parameterName2]...)
{
   statements;
}
```

Note that inline functions can bring a heavy cost. If the function is called 10 times, the inline code is copied into the calling functions each of those 10 times. The tiny improvement in speed you might achieve is more than swamped by the increase in size of the executable program. What's the rule of thumb? If you have a small function, one or two

statements, it is a candidate for `inline`. When in doubt, though, leave it out. Listing 5.9 demonstrates an `inline` function.

Listing 5.9. Demonstrates an inline function.

```
1:   // Listing 5.9 - demonstrates inline functions
2:
3:   #include <iostream.h>
4:
5:   inline int Double(int);
6:
7:   int main()
8:   {
9:     int target;
10:
11:    cout << "Enter a number to work with: ";
12:    cin >> target;
13:    cout << "\n";
14:
15:    target = Double(target);
16:    cout << "Target: " << target << endl;
17:
18:    target = Double(target);
19:    cout << "Target: " << target << endl;
20:
21:
22:    target = Double(target);
23:    cout << "Target: " << target << endl;
24:       return 0;
25:  }
26:
27:  int Double(int target)
28:  {
29:    return 2*target;
30: }
 Output: Enter a number to work with: 20

 Target: 40
 Target: 80
 Target: 160
```

Analysis: On line 5, `Double()` is declared to be an inline function taking an `int` parameter and returning an `int`. The declaration is just like any other prototype except that the keyword `inline` is prepended just before the return value. This compiles into code that is the same as if you had written the following:

```
target = 2 * target;
```

everywhere you entered

```
target = Double(target);
```

By the time your program executes, the instructions are already in place, compiled into the OBJ file. This saves a jump in the execution of the code, at the cost of a larger program.

5.8. *Recursive Functions*

Most mathematical functions that we are familiar with are described by a simple formula. For example, we can convert temperatures from Fahrenheit to Celsius by applying the formula

$$C = 5 (F-32) / 9$$

Given this formula, it is trivial to write a **C++** function; with declarations and braces removed, the one line formula above translates into one line C++ statement. Mathematical functions are sometimes defined in a less standard form. For example we can define a function f, valid on non negative integers, that satisfies:

$f(0) = 0$

$f(x) = 2, f(x-1) + x^2$ for $x>0$, $x \varepsilon Z$

From this definition,

$f(3) = 2f(2) + 3^2$

$= 2f(2) + 9$ now we should compute f(2) first.

$f(2) = 2f(1) + 2^2$

$= 2f(1) + 4$ now we should compute f(1) first.

$f(1) = 2f(0) + 1^2$

$= 2f(0) + 1$ now we should compute f(0) first.

$f(0) = 0$

⇨ $f(1) = 2 (0) + 1$

$= 0 + 1 = 1$

⇨ $f(2) = 2 (1) + 4$

$= 2 + 4 = 6$

=> $f(3) = 2 (6) + 9$

$= 12 + 9 = 21$

A function that is defined in terms of it self is called recursive. f(0) in the above function is the value which is the function directly known with out resorting to recursion. Such case is known as the base case. A recursive function without a base case is meaningless. When writing recursive routines, it is crucial to keep in mind the four basic rules of recursion. These are:

1. **Base case**: you must always have some Base case, which can be solved with out recursion.

2. **Making progress**: for the cases that are to be solved recursively, the recursive call must always be to a case that makes progress to ward a base case.
3. **Design rule**: Assume that all recursive calls work without fail. In order to find f(n) recursively you can assume f(n-k) will work ($k \geq 1$)
4. **Compound interest rule**: never duplicate work by solving the same instance of a problem in separate recursive rule.

Some examples of recursive problems

a) The factorial function f(n) = n!

f(n) can be defined as:

f(N) = N x (N-1) x (N-2) x (N-3) x ……..x 1 this is iterative defn

$$f(N) = \begin{cases} N \text{x } f(N-1) & \text{for } N > 0 \\ 1 & \text{for } N = 0 \end{cases}$$ this is called recursion.

This function can be implemented both iteratively or recursively as

```
//iterative implementation
int factorial (int N)
{
        int fact =1;
        for(int i=1;i<=n; i++)
        {
                fact *=I;
        }
        return fact;
}
```

```
//recursive implementation
int factorial (int n)
{
        if (n == 0) return 1;
        return n*factorial(n-1)
}
```

b) Fibonacci progress :

is a sequence in which the i^{th} term equals f(i) defined as

$$f(i) = \begin{cases} f(i-1) + f(i-2) & \text{for } i > 2 \\ 1 & \text{for } i = 1 \text{ and } 2 \end{cases}$$

Some of the elements in the sequence are

1, 1, 2, 3, 5, 8, 13, 21, 34 …….

The function can be implemented both iteratively and recursively. The recursive implementation is shown bellow.

```
int i_th_fibonacci_element (int n)
{
```

```
        if (n == 1|| n==2) return 1;
        else if  n > 2
        {
                int val1 =  i_th_fibonacci_element ( n-1);
                int val2 =  i_th_fibonacci_element ( n-2);
                return (val1 + val2);
        }
}
```

Some function cannot be implemented easily without recursion. Other functions, which do admit both iterative and recursive solution are easier to understand in their recursive form. So recursion is an essential tool for computer scientist.

However recursion is very expensive in terms of time complexity of a program so that it is not generally advisable to use recursion to problems that is simple to implement in iterative way. This is because of the time requirement to manipulate stack at each function call.