Chapter 1
Functions

1.1. Introduction to Modular Programming (Functions)

- 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.

- 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.

- Functions come in two varieties: user-defined and built-in.
- Built-in (Predefined) functions are part of your compiler package--they are supplied by the manufacturer for your use.
- A user-defined function are custom made functions that groups code to perform a specific task and that group of code is given a name (identifier).
- In this chapter we will discuss about user-defined functions.

Declaring and Defining Functions

- Using functions in your program requires that you first declare the function and 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**.

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.

Using separate prototype file

Myhe.h

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

void hello(string n){
    cout<<"hello"<<n;
}
```

Hellowrold.cpp

```
#include <iostream>
#include "myhe.h"
using namespace std;
int main(){
    string name;

    cin>>name;
    hello(name);
    return 0;
}
```

- 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);
- The same function with named parameters might be

  long Area(int length, int width);

- 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().

Defining the Function

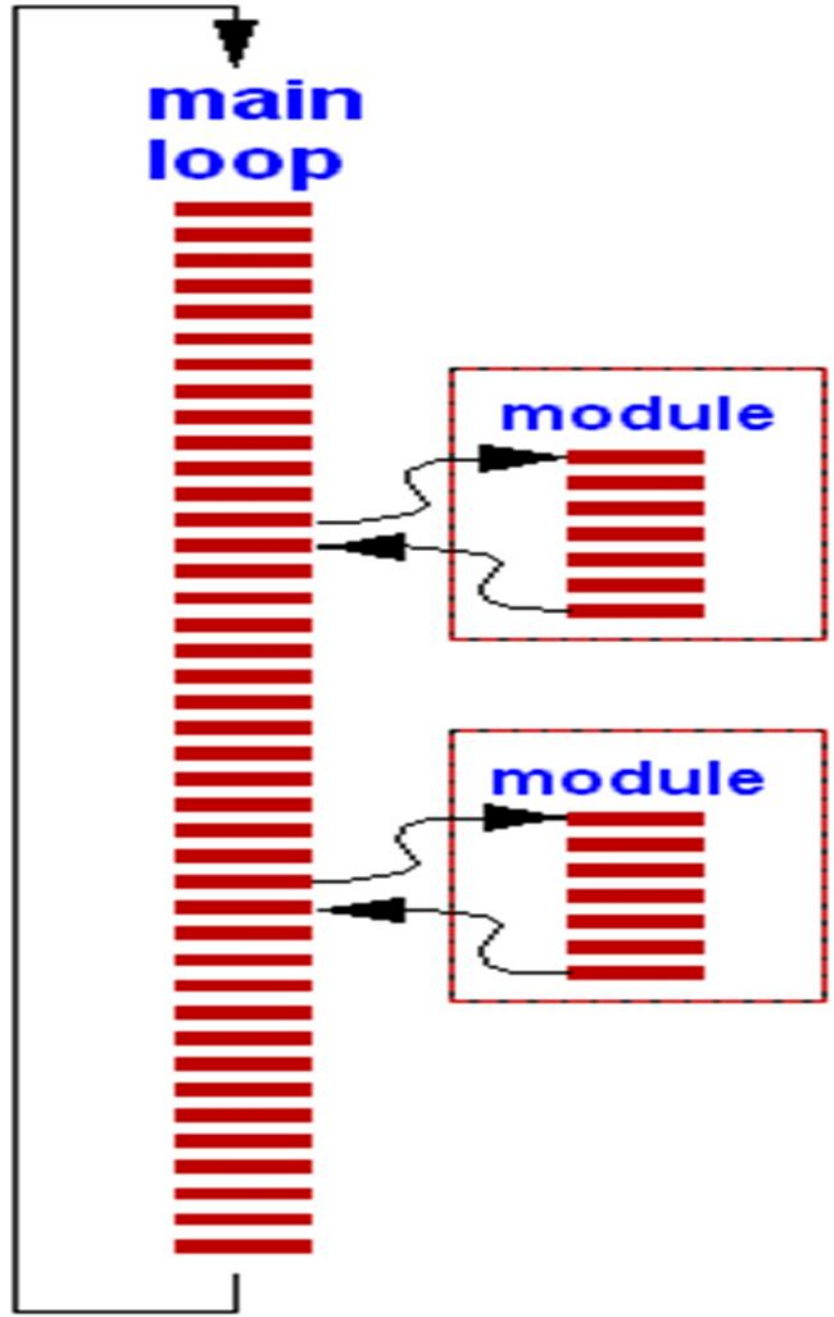
- The definition of a function consists of the function header and its body.
- The header is exactly like the function prototype, except that the parameters must be named, and there is no terminating semicolon.
- The body of the function is a set of statements enclosed in braces. Function Definition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

- 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.


main
loop
module
module

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.

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.

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:   }
```

```
17:   float Convert(float TFer)
18:   {
19:      float TCel;
20:      TCel = ((TFer - 32) * 5) / 9;
21:      return TCel;
22: }
```

Global Variables

- Global variables have global scope and are available anywhere within your program.
- 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.

```
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: }
```

```
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: }
```

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:  }
```

```
19:  void myFunc()
20:  {
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: }
```

Scope resolution operator

- When a local variable has the same name as a global variable, all references indicate to the variable name made within the scope of the local variable.

```
# 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 over 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.

```
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

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 longs, 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)))));

- 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

Passing arguments
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.

```
1:
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:     }
```

C

```
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: }
```

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.

```
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: }
```

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; //returns the interger 5
  - return (x > 5);// will be zero if x is not greater than 5, or it will be 1.
  - return (MyFunction());//MyFunction() itself returns a value

- 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.

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

```
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:  }
```

```
26:  AreaCube(int length, int width, int height)
27:  {
28:
29:      return (length * width * height);
30:  }
```

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.**

```
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: }
```

```
27: int Double(int target)
28: {
29:   return 2*target;
30: }
```

Recursive Functions

- A function that is defined in terms of it self is called recursive.
- 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 >=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
  - The factorial function f(n) = n!
  - 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.