Chapter: 9

# Introduction to C++

OOP, C++ console I/O, class intro., function overloading, C and C++ differences

## 9.1 Get familiar with OOP (Object Oriented Programming)

By using a ***high-level*** language, a programmer was able to write programs that were several thousand lines long. However, the method of programming used early on was an ad hoc, anything-goes approach. While this is fine for relatively short programs, it yields unreadable (and unmanageable) "***spaghetti code***" when applied to larger programs. ***Structured programming*** then solves this problem.

- ***Structured programming*** relies on well-defined ***control structures***, ***code blocks***, the ***absence*** (or at least minimal use) of the ***GOTO***, and ***stand-alone subroutines*** that support ***recursion*** and ***local variables***. The essence of *structured programming* is the *reduction of a program into its constituent elements*. Using structured programming, the average programmer can create and maintain programs that are up to 50,000 lines long. Although structured programming has yielded excellent results when applied to moderately complex programs, even it fails at some point, after a program reaches a certain size : the problems were about data ***abstraction***, code ***reusability*** and ***maintenance***.
- To allow more complex programs to be written ***object-oriented programming*** was invented. ***OOP*** takes the *best of the ideas embodied in structured programming and combines them with powerful new concepts* that allow us to *organize our programs* more effectively. OOP lets us to ***decompose*** a problem into its constituent parts. Each component becomes a ***self-contained object*** that contains its own ***instructions*** and ***data*** that relate to that object. In this way, complexity is reduced and the programmer can manage larger programs.
- All OOP languages, including C++, share three common defining traits: ***encapsulation***, ***polymorphism*** and ***inheritance***
  - ***ENCAPSULATION :*** Encapsulation is the mechanism that binds code and the data together, and keeps both safe from outside interference and misuse like a self-contained "black box". Linking code and data together in this fashion, creates an object.
    - Within an ***object***, code, data, or both may be ***private*** to that ***object*** or public.
    - ***Private*** code or data is known to and accessible only by another part of the object. That is, ***private*** code or data cannot be accessed by a piece of the program that exists outside the object.
    - When code or data is ***public***, other parts of the program can access it even though it is defined within an object.
    - Typically, the ***public*** parts of an object are used *to provide a controlled interface to the private elements of the object*.
    - For all intents and purposes, an *object is a variable of a user-defined type (more like a structure type variables. In C++ classes are more like structures and objects are more like variables ).*
  - ***POLYMORPHISM :*** "***To morph***" means "***to change***". So "$Morph = shape\ shifting$" and "*Morphism = shape shifting property*". ***Polymorphism*** (from the Greek, meaning "***many forms***") is the quality that allows one name to be used for two or more *related but technically different* purposes. As it relates to OOP, *polymorphism allows one name to specify a general class of actions*. Within a general class of actions, the specific action to be applied is determined by the type of data. It allows us to handle greater complexity by allowing the creation of standard interfaces to related activities.

    ***For example***, in ***C***, the ***absolute value*** action requires ***three*** distinct ***function*** names: `abs()`, `labs()`, and `fabs()`. These functions compute and return the absolute value of an ***integer***, a ***long integer***, and a ***floating-point*** value, respectively. However, in C++, which supports polymorphism, each function can be called by the same name, such as `abs()`.
    *[we'll discuss it later in "function overloading". In C++, it is possible to use one function name for many different purposes. This is called function overloading.].*
    - The type of data used to call the function determines which specific version of the function is actually executed.
    - More generally, the concept of polymorphism is characterized by the idea of "***one interface, multiple methods***," which means using a ***generic interface*** for a group of ***related activities***.
    - Polymorphism helps to reduce complexity by allowing one interface to specify a general class of action. (It is the compiler's job to select the specific action as it applies to each situation.) As in previous example, having three names for the absolute value function instead of just one makes the general activity of obtaining the absolute value more complex than actul.
    - ***Operator overloading :*** Polymorphism can be applied to operators, too. Virtually all programming languages contain a limited application of polymorphism as it relates to the arithmetic operators. For example, in C, the **+** sign is used to add ***integers***, ***long integers***, ***characters***, and ***floating-point*** values. In these cases, the compiler automatically knows which type of arithmetic to apply. In C++, this concept is extend for other types of data and it is called ***operator overloading***.
  - ***INHERITANCE :*** Inheritance is the process by which one object can acquire the properties of another. More specifically, an object can inherit a general set of properties to which it can add those features that are specific only to itself.
    - Inheritance is important because it allows an object to support the concept of hierarchical classification. Most information is made manageable by hierarchical classification. (More like set and their subsets). For example, think about the description of *a house*. A house is a part of the general class called *building*. In turn, building is part of the more general class *structures*. In each case, the child class inherits all those qualities associated with the parent and adds to them its own defining characteristics.
    - Through inheritance, it is possible to describe an object by stating what general class (or classes) it belongs to along with those specific traits that make it unique.

## 9.2 Old Header And Standard Header Declaration Of C++

The differences between ***old-style*** and ***modern code*** involve two new features: ***new-style headers*** and the ***namespace statement***. The first version, shown here, reflects the ***old-style coding***.

```
/*     A traditional - style C++ program .     */
#include <iostream.h>
int main(){
            /* program code */
      return 0;}
```

Pay special attention to the ***#include*** statement. This statement includes the file ***iostream.h***, which provides support for C++'s ***I/O system***. (It is to C++ what ***stdio.h*** is to C.) Following is the second version of the skeleton, which uses the *modern style*:

```
/*   A modern - style C++ program that uses the new - style headers and a namespace .  */
# include <iostream>
using namespace std;
int main(){
            /* program code */
      return 0;}
```

Notice the two lines in this program immediately after the first comment; this is where the changes occur. First, in the ***#include*** statement, there is no ***.h*** after the name ***iostream***. And second, the next line, specifying a ***namespaces***.

***The new header :*** When you use a library function in a program, you must ***include*** its ***header*** file. This is done using the ***#include*** statement. For example, in C we used ***#include <stdio.h>*** , to include the *header file* ***stdio.h*** for the ***I/O*** functions. Here ***stdio.h*** is the name of the file used by the ***I/O*** functions, and the ***#include <stdio.h>*** statement cause that file to be ***included*** in your program. *The key point is that the* ***#include*** *statement includes a file*.

The ***new-style*** headers do not specify filenames. Instead, they simply specify ***standard identifiers*** *that might be mapped to files by the compiler, but they need not be*. The ***new-style*** C++ headers are abstractions that simply guarantee that the appropriate ***prototypes*** and ***definitions*** required by the C++ library have been ***declared***. Since the *new-style header is not a filename, it does not have a* ***".h"*** *extension*. For example, here are some of the new-style ***headers*** supported by ***Standard C++:***

***<iostream>,  <fstream>,  <vector>,  <string>***

The new-style headers are included using the #include statement.

- ***C++*** still supports the standard ***C-style*** header files such as ***stdio.h*** and ***ctype.h***. However, Standard C++ also defines ***new-style headers*** that can be used in place of these header files.
- The ***C++*** versions of the ***standard C headers*** simply add a ***c prefix*** to the filename and ***drop the .h***. For example, the new style C++ header for ***math.h*** is ***<cmath>***, and for ***string.h*** is ***<cstring>***.

***The namespace :*** When you include a ***new-style*** header in your program *the contents of that header are contained in the* ***" std namespace "***. A ***namespace*** is simply a ***declarative region***. The purpose of a namespace is to ***localize*** the names of ***identifiers*** to avoid ***name collisions***. Traditionally, the names of library functions and other such items were simply placed into the ***global namespace*** (as they are in C). However, the contents of ***new-style*** headers are placed in the ***std namespace***.

- To bring the ***std namespace*** into visibility (i.e., to put ***std*** into the ***global namespace***) just use ***" using namespace std; "***. After this statement has been compiled, there is no difference between working with an ***old-style*** header and a ***new-style*** one.

NOTE

[1] The traditional way to include the I/O header is ***#include <iostream.h>*** this causes the file ***iostream.h*** to be included in your program. In general, an ***old-style*** header will use the same name as its corresponding ***new-style*** header with a ***.h*** appended.

[2] To transforms a ***modern program*** into a ***traditional-style*** one just replace

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

with

```
#include <iostream.h>
```

Since the ***old style*** header reads all of its contents into the ***global namespace***, there is no need for a *namespace statement*.

## 9.3 Function overloading in C++

In C++ function overloading is one type of ***polymorphism*** (*compile-time polymorphism*). In C++, two or more functions can share the same name as long as either the ***type*** of their ***arguments differs*** or the ***number*** of their ***arguments*** differs-or both. When two or more functions share the same name, they are said to be ***overloaded***.

- Overloaded functions can help reduce the complexity of a program by allowing related operations to be referred to by the same name.
- It is very easy to overload a function: *simply declare and define all required versions*. The compiler will automatically select the correct version based upon the number and/or type of the arguments used to call the function.
- It is also possible in C++ to overload ***operators***.

☞ The ***return type*** alone is ***not a sufficient difference*** to allow function overloading. If two functions differ only in the type of data they return, the compiler will not always be able to select the proper one to call. For example, this fragment is incorrect because it is inherently ambiguous.

```
int f1(int a);
double f1( int a);
. . . . .
f1(10) ;     /* which function does the compiler call ???*/
```

As the comment indicates, the compiler has no way of knowing which version of ***f1()*** to call.

***EXAMPLES***

The classic example of this situation is found in the C standard library contains the functions ***abs(), labs(),*** and ***fabs()***, which return the absolute value of an ***integer***, a ***long integer***, and a ***floating-point*** value, respectively. In C++, you can overload one name for the three types of data, as this example illustrates:

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

/* Overloading three abs() functions with different types */
int abs(int n);
long abs(long n);
double abs(double n);

int main(){ cout<<"Absolute of -10 : "<<abs(-10)<<"\n";
            cout<<"Absolute of -10L : "<<abs(-10L)<<"\n";
            cout<<"Absolute of -10.01 : "<<abs(-10.01)<<"\n";}

/* abs() for ints */
int abs(int n){ cout<< "\n In integer abs() = ";
                return n<0 ? -n : n; }

/* abs() for longs */
long abs(long n){cout<< "\n In long abs() = ";
                 return n<0 ? -n : n; }

/* abs() for doubles */
double abs(double n){cout<< "\n In double abs() = ";
                     return n<0 ? -n : n; }
```

## 9.4 C++ comments

In C++, you can include comments in your program two different ways.

- First, you can use the standard, C-like comment mechanism. That is, begin a comment with ***/**** and end it with ****/***. As with C, this type of comment cannot be nested in C++.
- The second way that you can add a remark to your C++ program is to use the single-line comment. A single-line comment begins with a ***//*** and stops at the end of the line.
- Typically, C++ programmers use C-like comments for multiline commentaries and reserve C++ style single line comments for short remarks.

## 9.5 C++ Console I/O

We may still use functions such as printf() and scanf() these are I/O functions. In C++, I/O is performed using I/O operators instead of I/O functions. In order to use the C++ I/O operators, you must include the header <iostream> in your program.

***Output:*** The *output operator* is **<<** . In general, to output to the console, use this form of the **<<** operator:

***cout << expression ;***

Here, ***expression*** can be any valid C++ expression-including ***another output*** expression. ***cout*** is a *predefined stream* that is automatically linked to the console when a C++ program begins execution. It is similar to C's ***stdout***.

☞ By using the **<<** output operator, it is possible to output any of C++'s ***basic types***. For example, this statement outputs the value 100.99:

```
cout << 100.99;
cout << " This string is output to the screen .\n";
```

***Input:*** To input a value from the keyboard, use the **>>** *input operator*. In general, to ***input*** values from the keyboard, use the form if **>>** :

***cin >> variable ;***

For example, `int num; cin >> num;` inputs an ***integer*** value into ***num***. ***Notice*** that ***num*** is not preceded by an **&**. ***When you use C's scanf() function to input values, variables must have their addresses passed to the function so they can receive the values entered by the user***. This is not the case when you are using C++'s input operator.

NOTE

[1] As you know, in C, these are the left and right shift operators, respectively. In C++, they still retain their original meanings (***left shift*** and ***right shift***) butt they also take on the expanded role of performing input and output.

[2] As in C, C++ ***console I/O*** may be redirected, but for the rest of this discussion, it is assumed that the console is being used.

## 9.6 Difference between C and C++

- ***Void is not essential for empty parameter :*** In C, when a function takes ***no parameters***, its ***prototype*** has the word ***void*** inside its *function parameter list*. For example, in C, if a function called ***f1()*** takes ***no parameters*** (and returns a ***char***), its prototype will look like this: `char f1(void);`
  However, in C++, the ***void*** is ***optional***. Therefore, in C++, the prototype for ***f1()*** is usually written like this: `char f1();`
  - In a C program, it is common practice to declare main() as shown here if it takes *no command-line arguments*:
    `int main(void)`
    However, in C++, the use of ***void*** is ***redundant*** and ***unnecessary*** and we'll use : `int main()`
  - in a C program, `char f1();` simply mean that *nothing is said about the parameters to the function* and `char f1(void);` means that the *function takes no parameters*.
  - In C++, it means that the function has no parameters. In C++, these two declarations are equivalent:
    `char f1();` ⇔ `char f1(void);`

- ***Prototype is essential :*** In a C++ program, all functions must be prototyped. ***C++ programs will not compile if the function is not prototyped***. [Remember, in C, prototypes are recommended but technically optional.]
  - A ***member*** function's ***prototype*** contained in a ***class*** (we'll discuss about class in next chapter) also serves as its ***general prototype***, and no other separate prototype is required.

- ***returning values are essential :*** In C++, if a function is declared as returning a value, it must return a value. We must explicitly declare the return type of all functions.
  - In C, a non-void function is not required to actually return a value. If it doesn't, a garbage value is "returned". In C, if you don't explicitly specify the return type of a function, an integer return type is assumed.

- ***Local variable anywhere :*** In C++, ***local*** variables can be declared ***anywhere***. One advantage of this approach is that ***local*** variables can be declared close to where they are first used, thus helping to prevent unwanted side effects.
  - In C, local variables can declared only at the ***start*** of a ***block***, prior to any ***"action"*** statements.

- ***New "bool" data-type :*** C++ defines the ***bool*** data type, which is used to store ***Boolean*** (i.e., ***true***/***false***) values. C++ also defines the *keywords* ***true*** and ***false***, which are the only values that a value of type ***bool*** can have.
  - In C++, the outcome of the ***relational*** and ***logical*** operators is a value of type ***bool***, and all conditional statements must evaluate to a ***bool*** value.
  - In C, ***true*** is any ***nonzero*** value and ***false*** is **0**. This still holds in C++ because any ***nonzero*** value is *automatically converted* into ***false*** when used in a *Boolean expression*. The reverse also occurs: ***true*** is converted to **1** and ***false*** is converted to **0** when a ***bool*** value is used in an *integer expression*.

- Here are a few more that you should be aware of:
  - In C, a character constant is automatically elevated to an integer, whereas in C++, it is not.
  - In C, it is not an error to declare a global variable several times, even though it is a bad programming practice. In C++, this is an error.
  - In C, an identifier will have at least 31 significant characters. In C++, all characters are considered significant. However, from a practical point of view, extremely long identifiers are unwieldy and are seldom needed.
  - In C, you can call ***main()*** from within a program, although this would be unusual. In C++, this is not allowed.
  - In C, you cannot take the address of a ***register*** variable. In C++, you can.
  - In C, the type ***wchar_t*** is defined with a ***typedef***. In C++, ***wchar_t*** is a keyword.

## 9.7 C++ Keywords

C++ supports all 32 of the keywords defined by C and adds 30 of its own. The entire set of keywords defined by C++ is shown in ***table*** below. Also early versions of C++ ***defined*** the overload keyword, but it is now obsolete.

| ***C++ Keywords*** | | | | | | |
|---|---|---|---|---|---|---|
| asm | const_cast | explicit | int | register | switch | union |
| auto | continue | extern | long | reinterpret_cast | template | unsigned |
| bool | default | false | mutable | return | this | using |
| break | delete | float | namespace | short | throw | virtual |
| case | do | for | new | signed | true | void |
| catch | double | friend | operator | sizeof | try | volatile |
| char | dynamic_cast | goto | private | static | typedef | wchar_t |
| class | else | if | protected | static_cast | typeid | while |
| const | enum | inline | public | struct | typename | |