Basics of C++ (Part 1)

Why C++

C++ is a middle-level programming language developed by Bjarne Stroustrup starting in 1979 at Bell Labs. C++ runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.
This C++ tutorial adopts a simple and practical approach to describe the concepts of C++ for beginners to advanced software engineers.
C++ is a MUST for students and working professionals to become a great Software Engineer. I will list down some of the key advantages of learning C++:
• C++ is very close to hardware, so you get a chance to work at a low level which gives you lot of control in terms of memory management, better performance and finally a robust software development.
• C++ programming gives you a clear understanding about Object Oriented Programming. You will understand low level implementation of polymorphism when you will implement virtual tables and virtual table pointers, or dynamic type identification.
• C++ is one of the every green programming languages and loved by millions of software developers. If you are a great C++ programmer then you will never sit without work and more importantly you will get highly paid for your work.
• C++ is the most widely used programming languages in application and system programming. So you can choose your area of interest of software development.
• C++ really teaches you the difference between compiler, linker and loader, different data types, storage classes, variable types their scopes etc.
There are 1000s of good reasons to learn C++ Programming. But one thing for sure, to learn any programming language, not only C++, you just need to code, and code and finally code until you become expert.

Example of Hello World using C++:

```
#include < iostream >
using namespace std;
// main() is where program execution begins.
int main() {
cout << "Hello World"; // prints Hello World
return 0;
}
```

There are many C++ compilers available which you can use to compile and run above mentioned program:
• Apple C++. Xcode
• Bloodshed Dev-C++
• Clang C++
• Cygwin (GNU C++)
• Mentor Graphics
• MINGW - "Minimalist GNU for Windows"
• GNU CC source
• IBM C++
• Intel C++
• Microsoft Visual C++
• Oracle C++
• HP C++
It is really impossible to give a complete list of all the available compilers. The C++ world is just too large and too much new is happening.

Applications of C++ Programming

As mentioned before, C++ is one of the most widely used programming languages. It has it's presence in almost every area of software development. I'm going to list few of them here:
• Application Software Development - C++ programming has been used in developing almost all the major Operating Systems like Windows, Mac OSX and Linux. Apart from the operating systems, the core part of many browsers like Mozilla Firefox and Chrome have been written using C++. C++ also has been used in developing the most popular database system called MySQL.
• Programming Languages Development - C++ has been used extensively in developing new programming languages like C#, Java, JavaScript, Perl, UNIX’s C Shell, PHP and Python, and Verilog etc.
• Computation Programming - C++ is the best friends of scientists because of fast speed and computational efficiencies.
• Games Development - C++ is extremely fast which allows programmers to do procedural programming for CPU intensive functions and provides greater control over hardware, because of which it has been widely used in development of gaming engines.
• Embedded System - C++ is being heavily used in developing Medical and Engineering Applications like softwares for MRI machines, high-end CAD/CAM systems etc.
This list goes on, there are various areas where software developers are happily using C++ to provide great softwares. I highly recommend you to

learn C++ and contribute great softwares to the community.

C++ Overview

C++ is a statically typed, compiled, general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming.
C++ is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features.
C++ was developed by Bjarne Stroustrup starting in 1979 at Bell Labs in Murray Hill, New Jersey, as an enhancement to the C language and originally named C with Classes but later it was renamed C++ in 1983.
C++ is a superset of C, and that virtually any legal C program is a legal C++ program.
Note − A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time.

Object-Oriented Programming

C++ fully supports object-oriented programming, including the four pillars of object-oriented development-
• Encapsulation
• Data hiding
• Inheritance
• Polymorphism

Standard Libraries

Standard C++ consists of three important parts:
• The core language giving all the building blocks including variables, data types and literals, etc.
• The C++ Standard Library giving a rich set of functions manipulating files, strings, etc.
• The Standard Template Library (STL) giving a rich set of methods manipulating data structures, etc.

The ANSI Standard

The ANSI standard is an attempt to ensure that C++ is portable; that code you write for Microsoft's compiler will compile without errors, using a compiler on a Mac, UNIX, a Windows box, or an Alpha.
The ANSI standard has been stable for a while, and all the major C++ compiler manufacturers support the ANSI standard.

Learning C++

The most important thing while learning C++ is to focus on concepts.
The purpose of learning a programming language is to become a better programmer; that is, to become more effective at designing and implementing new systems and at maintaining old ones.
C++ supports a variety of programming styles. You can write in the style of Fortran, C, Smalltalk, etc., in any language. Each style can achieve its aims effectively while maintaining runtime and space efficiency.

Use of C++

C++ is used by hundreds of thousands of programmers in essentially every application domain.
C++ is being highly used to write device drivers and other software that rely on direct manipulation of hardware under realtime constraints.
C++ is widely used for teaching and research because it is clean enough for successful teaching of basic concepts.
Anyone who has used either an Apple Macintosh or a PC running Windows has indirectly used C++ because the primary user interfaces of these systems are written in C++.

C++ Environment Setup

Local Environment Setup

If you are still willing to set up your environment for C++, you need to have the following two softwares on your computer.

Text Editor

This will be used to type your program. Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.
Name and version of text editor can vary on different operating systems. For example, Notepad will be used on Windows and vim or vi can be used on windows as well as Linux, or UNIX.
The files you create with your editor are called source files and for C++ they typically are named with the extension .cpp, .cp, or .c.
A text editor should be in place to start your C++ programming.

C++ Compiler

This is an actual C++ compiler, which will be used to compile your source code into final executable program.

Most C++ compilers don't care what extension you give to your source code, but if you don't specify otherwise, many will use .cpp by default. Most frequently used and free available compiler is GNU C/C++ compiler, otherwise you can have compilers either from HP or Solaris if you have the respective Operating Systems.

Installing GNU C/C++ Compiler

UNIX/Linux Installation

If you are using Linux or UNIX then check whether GCC is installed on your system by entering the following command from the command line:

```
$ g++ -v
```

If you have installed GCC, then it should print a message such as the following:

```
Using built-in specs.
Target: i386-redhat-linux
Configured with: ../configure --prefix=/usr .......
Thread model: posix
gcc version 4.1.2 20080704 (Red Hat 4.1.2-46)
```

If GCC is not installed, then you will have to install it yourself using the detailed instructions available at https://gcc.gnu.org/install/

Mac OS X Installation

If you use Mac OS X, the easiest way to obtain GCC is to download the Xcode development environment from Apple's website and follow the simple installation instructions.
Xcode is currently available at developer.apple.com/technologies/tools/.

Windows Installation

To install GCC at Windows you need to install MinGW. To install MinGW, go to the MinGW homepage, www.mingw.org, and follow the link to the MinGW download page.
Download the latest version of the MinGW installation program which should be named MinGW-< version >.exe.
While installing MinGW, at a minimum, you must install gcc-core, gcc-g++, binutils, and the MinGW runtime, but you may wish to install more.
Add the bin subdirectory of your MinGW installation to your PATH environment variable so that you can specify these tools on the command line by their simple names.
When the installation is complete, you will be able to run gcc, g++, ar,

ranlib, dlltool, and several other GNU tools from the Windows command line.

C++ Basic syntax

When we consider a C++ program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what a class, object, methods, and instant variables mean.
• Object − Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors - wagging, barking, eating. An object is an instance of a class.
• Class − A class can be defined as a template/blueprint that describes the behaviors/states that object of its type support.
• Methods − A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
• Instance Variables − Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

Example of C++ Program Structure:

```
#include < iostream >
using namespace std;
// main() is where program execution begins.
int main()
cout << "Hello World"; // prints Hello World
return 0;
}
```

Let us look at the various parts of the above program:
• The C++ language defines several headers, which contain information that is either necessary or useful to your program. For this program, the header < iostream > is needed.
• The line using namespace std; tells the compiler to use the std namespace. Namespaces are a relatively recent addition to C++.
• The next line '// main() is where program execution begins.' is a single-line comment available in C++. Single-line comments begin with // and stop at the end of the line.
• The line int main() is the main function where program execution begins.
• The next line cout << "Hello World"; causes the message "Hello World" to be displayed on the screen.
• The next line return 0; terminates main( )function and causes it to return the value 0 to the calling process.

Compile and Execute C++ Program

Let's look at how to save the file, compile and run the program. Please follow the steps given below:

• Open a text editor and add the code as above.
• Save the file as: hello.cpp
• Open a command prompt and go to the directory where you saved the file.
• Type 'g++ hello.cpp' and press enter to compile your code. If there are no errors in your code the command prompt will take you to the next line and would generate a.out executable file.
• Now, type 'a.out' to run your program.
• You will be able to see ' Hello World ' printed on the window.

```
$ g++ hello.cpp
$ ./a.out
Hello World
```

Make sure that g++ is in your path and that you are running it in the directory containing file hello.cpp.
You can compile C/C++ programs using makefile.

Semicolons and Blocks in C++

In C++, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.
For example, following are three different statements −

```
x = y;
y = y + 1;
add(x, y);
```

A block is a set of logically connected statements that are surrounded by opening and closing braces. For example-

```
{
cout << "Hello World"; // prints Hello World
return 0;
}
```

C++ does not recognize the end of the line as a terminator. For this reason, it does not matter where you put a statement in a line. For example−

```
x = y;
y = y + 1;
add(x, y);
```

is the same as

```
x = y; y = y + 1; add(x, y);
```

C++ Identifiers

A C++ identifier is a name used to identify a variable, function, class, module, or any other user-defined item.
An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9).
C++ does not allow punctuation characters such as @, $, and % within identifiers. C++ is a case-sensitive programming language. Thus, Manpower and manpower are two different identifiers in C++.
Here are some examples of acceptable identifiers −
mohd zara abc move_name a_123
myname50 _temp j a23b9 retVal

C++ Keywords

The following list shows the reserved words in C++.
These reserved words may not be used as constant or variable or any other identifier names.
asm else new this
auto enum operator throw
bool explicit private true

Trigraphs

A few characters have an alternative representation, called a trigraph sequence.
A trigraph is a three-character sequence that represents a single character and the sequence always starts with two question marks.
Trigraphs are expanded anywhere they appear, including within string literals and character literals, in comments, and in preprocessor directives.
Following are most frequently used trigraph sequences −

= #
/ \
' ^
([
)]
! |
< {
> }
- ~
All the compilers do not support trigraphs and they are not advised to be used because of their confusing nature.

Whitespace in C++

A line containing only whitespace, possibly with a comment, is known as a blank line, and C++ compiler totally ignores it.
Whitespace is the term used in C++ to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins.
Statement 1:

```
int age;
```

In the above statement there must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them.
Statement 2

```
fruit = apples + oranges; // Get the total fruit
```

In the above statement 2, no whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish for readability purpose.

Comments in C++

Program comments are explanatory statements that you can include in the C++ code. These comments help anyone reading the source code. All programming languages allow for some form of comments.
C++ supports single-line and multi-line comments. All characters available inside any comment are ignored by C++ compiler.
C++ comments start with /* and end with */. For example −

```
/* This is a comment */
/* C++ comments can also
* span multiple lines
*/
```

A comment can also start with //, extending to the end of the line.

Example:

```
#include < iostream >
using namespace std;
main() {
cout << "Hello World"; // prints Hello World
return 0;
}
```

When the above code is compiled, it will ignore // prints Hello World and final executable will produce the following result −

```
Hello World
```

Within a /* and */ comment, // characters have no special meaning. Within a // comment, /* and */ have no special meaning. Thus, you can "nest" one kind of comment within the other kind. For example −

```
/* Comment out printing of Hello World:
cout << "Hello World"; // prints Hello World
*/
```

C++ Data types

While writing program in any language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.
You may like to store information of various data types like character, wide character, integer, floating point, double floating point, boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

Primitive Built-in Types

C++ offers the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types −
Boolean -- bool
Character -- char
Integer -- int
Floating point -- float Double floating point -- double
Valueless -- void
Wide character -- wchar_t
Several of the basic types can be modified using one or more of these type modifiers −signed
• unsigned
• short
• long

Example:

```
#include < iostream >
using namespace std;
int main() {
cout << "Size of char : " << sizeof(char) << endl;
cout << "Size of int : " << sizeof(int) << endl;
cout << "Size of short int : " << sizeof(short int) << endl;
cout << "Size of long int : " << sizeof(long int) << endl;
cout << "Size of float : " << sizeof(float) << endl;
cout << "Size of double : " << sizeof(double) << endl;
cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;
return 0;
}
```

This example uses endl, which inserts a new-line character after every line and << operator is being used to pass multiple values out to the screen. We are also using sizeof() operator to get size of various data types.
When the above code is compiled and executed, it produces the following result which can vary from machine to machine −

```
Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 4
Size of float : 4
Size of double : 8
Size of wchar_t : 4
```

Typedef Declarations

You can create a new name for an existing type using typedef. Following is the simple syntax to define a new type using typedef −

```
Typedef type newname;
```

For example, the following tells the compiler that feet is another name for int −

```
typedef int feet;
```

Now, the following declaration is perfectly legal and creates an integer variable called distance −

```
feet distance;
```

Enumerated Types

An enumerated type declares an optional type name and a set of zero or more identifiers that can be used as values of the type. Each enumerator is a constant whose type is the enumeration.
Creating an enumeration requires the use of the keyword enum. The general form of an enumeration type is −

```
enum enum-name { list of names } var-list;
```

Here, the enum-name is the enumeration's type name. The list of names is comma separated.
For example, the following code defines an enumeration of colors called colors and the variable c of type color. Finally, c is assigned the value "blue".

```
enum color { red, green, blue } c;
c = blue;
```

By default, the value of the first name is 0, the second name has the value

1, and the third has the value 2, and so on. But you can give a name, a specific value by adding an initializer. For example, in the following enumeration, green will have the value 5.
enum color { red, green = 5, blue };
Here, blue will have a value of 6 because each name will be one greater than the one that precedes it.

Character Streams

C++ VARIABLE TYPES A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.
The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C++ is case-sensitive −
There are following basic types of variable in C++ as explained in last chapter −

1 bool
Stores either value true or false.

2 char
Typically a single octet (one byte). This is an integer type.

3 int
The most natural size of integer for the machine.

4 float
A single-precision floating point value.

5 double
A double-precision floating point value.

6 void
Represents the absence of type.

7 wchar_t
A wide character type.

C++ also allows to define various other types of variables, which we will cover in subsequent chapters like Enumeration, Pointer, Array, Reference, Data structures, and Classes.
Following section will cover how to define, declare and use various types of

variables.

Variable Definition in C++

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type, and contains a list of one or more variables of that type as follows −
type variable_list;
Here, type must be a valid C++ data type including char, w_char, int, float, double, bool or any user-defined object, etc., and variable_list may consist of one or more identifier names separated by commas. Some valid declarations are shown here −
int i, j, k;
char c, ch;
float f, salary;
double d;
The line int i, j, k; both declares and defines the variables i, j and k; which instructs the compiler to create variables named i, j and k of type int.
Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows −
type variable_name = value;
Some examples are −
extern int d = 3, f = 5; // declaration of d and f.
int d = 3, f = 5; // definition and initializing d and f.
byte z = 22; // definition and initializes z.
char x = 'x'; // the variable x has the value 'x'.
For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables is undefined.

Variable Declaration in C++

A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable.
A variable declaration has its meaning at the time of compilation only, compiler needs actual variable definition at the time of linking of the program.
A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program. You will use extern keyword to declare a variable at any place.
Though you can declare a variable multiple times in your C++ program, but

it can be defined only once in a file, a function or a block of code.

Example:

```
#include < iostream >
using namespace std;
// Variable declaration:
extern int a, b;
extern int c;
extern float f;
int main () {
// Variable definition:
int a, b;
int c;
float f;
// actual initialization
a = 10;
b = 20;
c = a + b;
cout << c << endl ;
f = 70.0/3.0;
cout << f << endl ;
return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
30
23.3333
```

Same concept applies on function declaration where you provide a function name at the time of its declaration and its actual definition can be given anywhere else. For example −

```
// function declaration
int func();
int main() {
// function call
int i = func();
}
// function definition
int func() {
return 0;
}
```

Lvalues and Rvalues

There are two kinds of expressions in C++

- lvalue − Expressions that refer to a memory location is called "lvalue" expression. An lvalue may appear as either the left-hand or right-hand side of an assignment.
- rvalue − The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and can not appear on the left-hand side. Following is a valid statement −

```
int g = 20;
```

But the following is not a valid statement and would generate compile-time error −

```
10 = 20;
```

Basics of C++ (Part 2)

C++ Modifier types

C++ allows the char, int, and double data types to have modifiers preceding them. A modifier is used to alter the meaning of the base type so that it more precisely fits the needs of various situations.
The data type modifiers are listed here −

- signed
- unsigned
- long
- short

The modifiers signed, unsigned, long, and short can be applied to integer base types. In addition, signed and unsigned can be applied to char, and long can be applied to double.
The modifiers signed and unsigned can also be used as prefix to long or short modifiers. For example, unsigned long int.
C++ allows a shorthand notation for declaring unsigned, short, or long integers. You can simply use the word unsigned, short, or long, without int. It automatically implies int. For example, the following two statements both declare unsigned integer variables.

```
unsigned x;
unsigned int y;
```

Example:

```
#include < iostream >
using namespace std;
int main() {
short int i; // a signed short integer
short unsigned int j; // an unsigned short integer
j = 50000;
i = j;
cout << i << " " << j;
return 0;
}
```

When this program is run, following is the output −

```
-15536 50000
```

The above result is because the bit pattern that represents 50,000 as a short unsigned integer is interpreted as -15,536 by a short.

Type Qualifiers in C++

The type qualifiers provide additional information about the variables they precede.
1 const
Objects of type const cannot be changed by your program during execution.
2 volatile
The modifier volatile tells the compiler that a variable's value may be changed in ways not explicitly specified by the program.
3 restrict
A pointer qualified by restrict is initially the only means by which the object it points to can be accessed. Only C99 adds a new type qualifier called restrict.

Storage classes in C++

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program. These specifiers precede the type that they modify.
There are following storage classes, which can be used in a C++ Program • auto
• register
• static
• extern
• mutable

Example:

```
#include < iostream >
void func(void);
static int count = 10; /* Global variable */
main() {
while(count--) {
func();
}
return 0;
}
// Function definition
void func( void ) {
static int i = 5; // local static variable
i++;
std::cout << "i is " << i ;
std::cout << " and count is " << count << std::endl;
}
```

Operators in C++

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provide the following types

Arithmetic Operators

There are following arithmetic operators supported by C++ language −
Assume variable A holds 10 and variable B holds 20, then −

+ Adds two operands
- Subtracts second operand from the first
* Multiplies both operands
/ Divides numerator by de-numerator
% Modulus Operator and remainder of after an integer division
++ Increment operator, increases integer value by one
-- Decrement operator, decreases integer value by one

Relational Operators

There are following relational operators supported by C++ language
Assume variable A holds 10 and variable B holds 20, then −
== Checks if the values of two operands are equal or not, if yes then condition becomes true.
!= Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
> Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
< Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>= Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<= Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

Logical Operators

There are following logical operators supported by C++ language.
Assume variable A holds 1 and variable B holds 0, then −
&& Called Logical AND operator. If both the operands are non-zero, then condition becomes true.
|| Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.

! Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation.
Assume if A = 60; and B = 13; now in binary format they will be as follows −
A = 0011 1100
B = 0000 1101

A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A = 1100 0011

The Bitwise operators supported by C++ language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then −

& Binary AND Operator copies a bit to the result if it exists in both operands.
| Binary OR Operator copies a bit if it exists in either operand.
^ Binary XOR Operator copies the bit if it is set in one operand but not both.
~ Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<< Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>> Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

Assignment Operators

There are following assignment operators supported by C++ language −
= Simple assignment operator, Assigns values from right side operands to left side operand.
+= Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.
-= Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.
*= Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.
/= Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.
%= Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.

<<= Left shift AND assignment operator.
>>= Right shift AND assignment operator.
&= Bitwise AND assignment operator.
^= Bitwise exclusive OR and assignment operator.
|= Bitwise inclusive OR and assignment operator.

Misc Operators

The following table lists some other operators that C++ supports.

1 sizeof
sizeof operator returns the size of a variable. For example, sizeof(a), where 'a' is integer, and will return 4.
2 Condition ? X : Y
Conditional operator (?). If Condition is true then it returns value of X otherwise returns value of Y.
3 ,
Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
4 . (dot) and -> (arrow)
Member operators are used to reference individual members of classes, structures, and unions.
5 Cast
Casting operators convert one data type to another. For example, int(2.2000) would return 2.
6 &
Pointer operator & returns the address of a variable. For example &a; will give actual address of the variable.
7 *
Pointer operator * is pointer to a variable. For example *var; will pointer to a variable var.

Operators Precedence in C++

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator −
For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Postfix () [] -> . ++ - - Left to right
Unary + - ! ~ ++ - - (type)* & sizeof Right to left
Multiplicative * / % Left to right
Additive + - Left to right
Shift << >> Left to right
Relational < <= > >= Left to right
Equality == != Left to right
Bitwise AND & Left to right
Bitwise XOR ^ Left to right
Bitwise OR | Left to right
Logical AND && Left to right
Logical OR || Left to right
Conditional ?: Right to left
Assignment = += -= *= /= %=>>= <<= &= ^= |= Right to left
Comma , Left to right
Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

C++ Loops

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.
Programming languages provide various control structures that allow for more complicated execution paths.
A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages

1 while loop
Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2 for loop
Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3 do...while loop
Like a ‘while’ statement, except that it tests the condition at the end of the loop body.
4 nested loops
You can use one or more loop inside any another ‘while’, ‘for’ or ‘do..while’ loop.
C++ programming language provides the following type of loops to handle looping requirements.

Loop Control Statements

Loop control statements change execution from its normal sequence.
When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
C++ supports the following control statements.
1 break statement
Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
2 continue statement
Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3 goto statement
Transfers control to the labeled statement. Though it is not advised to use goto statement in your program.

The Infinite Loop

A loop becomes infinite loop if a condition never becomes false. The for loop is traditionally used for this purpose. Since none of the three expressions that form the ‘for’ loop are required, you can make an endless loop by leaving the conditional expression empty.
When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C++ programmers more commonly use the ‘for (;;)’ construct to signify an infinite loop.
NOTE − You can terminate an infinite loop by pressing Ctrl + C keys.

C++ Function

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is main(), and all the most trivial programs can define additional functions.
You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.
A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.
The C++ standard library provides numerous built-in functions that your program can call. For example, function strcat() to concatenate two strings, function memcpy() to copy one memory location to another location and many more functions.
A function is known with various names like a method or a sub-routine or a procedure etc.

Defining a Function

The general form of a C++ function definition is as follows −

```
return_type function_name( parameter list )
{
body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function −

- Return Type − A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.
- Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- 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. Parameters are optional; that is, a function may contain no parameters.
- Function Body − The function body contains a collection of statements that define what the function does.

Example:
Following is the source code for a function called max(). This function takes two parameters num1 and num2 and return the biggest of both −

```
// function returning the max between two numbers
int max(int num1, int num2) {
// local variable declaration
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
```

Function Declarations

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.
A function declaration has the following parts −

```
return_type function_name( parameter list );
```

For the above defined function max(), following is the function declaration −
int max(int num1, int num2);
Parameter names are not important in function declaration only their type is required, so following is also valid declaration −
int max(int, int);
Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.
When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it's return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.
To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value.

Example:

```
#include < iostream >
using namespace std;
// function declaration
int max(int num1, int num2);
int main () {
// local variable declaration:
int a = 100;
int b = 200;
int ret;
// calling a function to get max value.
ret = max(a, b);
cout << "Max value is : " << ret << endl;
return 0;
}
```

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result −

```
Max value is : 200
```

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.
The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
While calling a function, there are two ways that arguments can be passed to a function −

1 Call by Value
This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2 Call by Pointer
This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
3 Call by Reference
This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

Default Values for Parameters

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.
This is done by using the assignment operator and assigning values for the arguments in the function definition.
If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example –

Example:

```
#include < iostream >
using namespace std;
int sum(int a, int b = 20) {
```

```
int result;
result = a + b;
return (result);
}
int main () {
// local variable declaration:
int a = 100;
int b = 200;
int result;
// calling a function to add the values.
result = sum(a, b);
cout << "Total value is :" << result << endl;
// calling a function again as follows.
result = sum(a);
cout << "Total value is :" << result << endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Total value is: 300
Total value is: 120
```

C++ Array

C++ provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.
All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows −

```
type arrayName [ arraySize ];
```

This is called a single-dimension array. The arraySize must be an integer

constant greater than zero and type can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement −
double balance[10];

Initializing Arrays

You can initialize C++ array elements either one by one or using a single statement as follows −
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array −
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
You will create exactly the same array as you did in the previous example.
balance[4] = 50.0;
The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index.

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −
double salary = balance[9];
The above statement will take 10th element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays −

Example:

```
#include < iostream >
using namespace std;
int main () {
int n[ 10 ]; // n is an array of 10 integers
// initialize elements of array n to 0
for ( int i = 0; i < 10; i++ ) {
n[ i ] = i + 100; // set element at location i to i + 100
}
cout << "Element" << setw( 13 ) << "Value" << endl;
// output each array element's value
for ( int j = 0; j < 10; j++ ) {
cout << setw( 7 )<< j << setw( 13 ) << n[ j ] << endl;
}
return 0;
}
```

This program makes use of setw() function to format the output. When the above code is compiled and executed, it produces the following result −

```
Element Value
0 100
1 101
2 102
3 103
4 104
5 105
6 106
7 107
8 108
9 109
```

Arrays in C++

Arrays are important to C++ and should need lots of more detail. There are following few important concepts, which should be clear to a C++ programmer −

1 Multi-dimensional arrays
C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
2 Pointer to an array
You can generate a pointer to the first element of an array by simply specifying the array name, without any index.
3 Passing arrays to functions
You can pass to the function a pointer to an array by specifying the array's name without an index.
4 Return array from functions
C++ allows a function to return an array.

C++ Strings

C++ provides following two types of string representations −

- The C-style character string.
- The string class type introduced with Standard C++.

The C-Style Character String

The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
If you follow the rule of array initialization, then you can write the above statement as follows −
char greeting[] = "Hello";
Following is the memory presentation of above defined string in C/C++ −
Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string −

```
#include < iostream >
using namespace std;
int main () {
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
cout << "Greeting message: ";
cout << greeting << endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Greeting message: Hello
```

C++ supports a wide range of functions that manipulate null-terminated strings −
1 strcpy(s1, s2);
Copies string s2 into string s1.
2 strcat(s1, s2);
Concatenates string s2 onto the end of string s1.
3 strlen(s1);
Returns the length of string s1.
4 strcmp(s1, s2);
Returns 0 if s1 and s2 are the same; less than 0 if s1< s2; greater than 0 if s1>s2.
5 strchr(s1, ch);
Returns a pointer to the first occurrence of character ch in string s1.

6 strstr(s1, s2);
Returns a pointer to the first occurrence of string s2 in string s1.

Example:

```
#include < iostream >
#include < cstring > using namespace std;
int main () {
char str1[10] = "Hello";
char str2[10] = "World";
char str3[10];
int len ;
// copy str1 into str3
strcpy( str3, str1);
cout << "strcpy( str3, str1) : " << str3 << endl;
// concatenates str1 and str2
strcat( str1, str2);
cout << "strcat( str1, str2): " << str1 << endl;
// total lenghth of str1 after concatenation
len = strlen(str1);
cout << "strlen(str1) : " << len << endl;
return 0;
}
```

When the above code is compiled and executed, it produces result something as follows −

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

The String Class in C++

The standard C++ library provides a string class type that supports all the operations mentioned above, additionally much more functionality.

Example:

```
#include < iostream >
#include < cstring > using namespace std;
int main () {
string str1 = "Hello";
string str2 = "World";
string str3;
int len ;
// copy str1 into str3
str3 = str1;
cout << "str3 : " << str3 << endl;
// concatenates str1 and str2
str3 = str1 + str2;
cout << "str1 + str2 : " << str3 << endl;
// total length of str3 after concatenation
len = str3.size();
cout << "str3.size() : " << len << endl;
return 0;
}
```

C++ Pointers

C++ pointers are easy and fun to learn. Some C++ tasks are performed more easily with pointers, and other C++ tasks, such as dynamic memory allocation, cannot be performed without them.
As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory.
Consider the following which will print the address of the variables defined −

Example:

```
#include < iostream >
using namespace std;
int main () {
int var1;
char var2[10];
cout << "Address of var1 variable: ";
cout << &var1 << endl;
cout << "Address of var2 variable: ";
cout << &var2 << endl;
return 0;
}
```

```
Address of var1 variable: 0xbfebd5c0
Address of var2 variable: 0xbfebd5b6
```

What are Pointers?

A pointer is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is −

```
type *var-name;
```

Here, type is the pointer's base type; it must be a valid C++ type and var-name is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication.
However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration −

```
int *ip; // pointer to an integer
double *dp; // pointer to a double
float *fp; // pointer to a float
char *ch // pointer to character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address.
The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

Using Pointers in C++

There are few important operations, which we will do with the pointers very frequently. (a) We define a pointer variable. (b) Assign the address of a variable to a pointer. (c) Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations −

Example:

```
#include < iostream >
using namespace std;
int main () {
int var = 20; // actual variable declaration.
int *ip; // pointer variable
ip = &var; // store address of var in pointer variable
cout << "Value of var variable: ";
cout << var << endl;
// print the address stored in ip pointer variable
cout << "Address stored in ip variable: ";
cout << ip << endl;
// access the value at the address available in pointer
cout << "Value of *ip variable: ";
cout << *ip << endl;
return 0;
}
```

When the above code is compiled and executed, it produces result something as follows −

```
Value of var variable: 20
Address stored in ip variable: 0xbfc601ac
Value of *ip variable: 20
```

Pointers in C++

Pointers have many but easy concepts and they are very important to C++ programming. There are following few important pointer concepts which should be clear to a C++ programmer −

Sr.No	Concept & Description
1	Null Pointers C++ supports null pointer, which is a constant with a value of zero defined in several standard libraries.
2	Pointer Arithmetic There are four arithmetic operators that can be used on pointers: ++, --, +, -
3	Pointers vs Arrays There is a close relationship between pointers and arrays.
4	Array of Pointers You can define arrays to hold a number of pointers.
5	Pointer to Pointer C++ allows you to have pointer on a pointer and so on.
6	Passing Pointers to Functions Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function.
7	Return Pointer from Functions C++ allows a function to return a pointer to local variable, static variable and dynamically allocated memory as well.

C++ Basic Input/Output

The C++ standard libraries provide an extensive set of input/output capabilities which we will see in subsequent chapters.
C++ I/O occurs in streams, which are sequences of bytes.
If bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called input operation and if bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc., this is called output operation.

I/O Library Header Files

There are following header files important to C++ programs −
1 < iostream >
This file defines the cin, cout, cerr and clog objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.
2 < iomanip >
This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as setw and setprecision.
3 < fstream >
This file declares services for user-controlled file processing. We will discuss about it in detail in File and Stream related chapter.

The Standard Output Stream (cout)

The predefined object cout is an instance of ostream class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The cout is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.
The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The << operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values.
The insertion operator << may be used more than once in a single statement as shown above and endl is used to add a new-line at the end of the line.

The Standard Input Stream (cin)

The predefined object cin is an instance of istream class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The cin is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.
The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables.
The stream extraction operator >> may be used more than once in a single statement. To request more than one datum you can use the following −
cin >> name >> age;
This will be equivalent to the following two statements −
cin >> name;

cin >> age;

The Standard Error Stream (cerr)

The predefined object cerr is an instance of ostream class. The cerr object is said to be attached to the standard error device, which is also a display screen but the object cerr is un-buffered and each stream insertion to cerr causes its output to appear immediately.
The cerr is also used in conjunction with the stream insertion operator.

C++ Data Structure

C/C++ arrays allow you to define variables that combine several data items of the same kind, but structure is another user defined data type which allows you to combine data items of different kinds.
Structures are used to represent a record, suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

- Title
- Author
- Subject
- Book ID

Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member, for your program. The format of the struct statement is this −

```
struct [structure tag] {
member definition;
member definition;
...
member definition;
} [one or more structure variables];
```

The structure tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition.
At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure −

```
struct Books {
char title[50];
char author[50];
char subject[100];
int book_id;
} book;
```

Accessing Structure Members

To access any member of a structure, we use the member access operator (.).
The member access operator is coded as a period between the structure variable name and the structure member that we wish to access.
You would use struct keyword to define variables of structure type.
Following is the example to explain usage of structure –

Example:

```
#include < iostream >
#include < cstring >
using namespace std;
struct Books {
char title[50];
char author[50];
char subject[100];
int book_id;
};
int main() {
struct Books Book1; // Declare Book1 of type Book
struct Books Book2; // Declare Book2 of type Book
// book 1 specification
strcpy( Book1.title, "Learn C++ Programming");
strcpy( Book1.author, "Chand Miyan");
strcpy( Book1.subject, "C++ Programming");
Book1.book_id = 6495407;
// book 2 specification
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Yakit Singha");
strcpy( Book2.subject, "Telecom");
Book2.book_id = 6495700;
// Print Book1 info
cout << "Book 1 title : " << Book1.title << endl;
cout << "Book 1 author : " << Book1.author << endl;
cout << "Book 1 subject : " << Book1.subject << endl;
cout << "Book 1 id : " << Book1.book_id << endl; // Print Book2 info
cout << "Book 2 title : " << Book2.title << endl;
cout << "Book 2 author : " << Book2.author << endl;
cout << "Book 2 subject : " << Book2.subject << endl;
cout << "Book 2 id : " << Book2.book_id << endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Book 1 title : Learn C++ Programming
Book 1 author : Chand Miyan
Book 1 subject : C++ Programming
Book 1 id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Yakit Singha
Book 2 subject : Telecom
Book 2 id : 6495700
```

Structures as Function Arguments

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the similar way as you have accessed in the above example –

Example:

```
#include < iostream >
#include < cstring >
using namespace std;
void printBook( struct Books book );
struct Books {
char title[50];
char author[50];
char subject[100];
int book_id;
};
int main() {
struct Books Book1; // Declare Book1 of type Book
struct Books Book2; // Declare Book2 of type Book
// book 1 specification
strcpy( Book1.title, "Learn C++ Programming");
strcpy( Book1.author, "Chand Miyan");
strcpy( Book1.subject, "C++ Programming");
Book1.book_id = 6495407;
// book 2 specification
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Yakit Singha");
strcpy( Book2.subject, "Telecom");
Book2.book_id = 6495700;
// Print Book1 info
printBook( Book1 );
// Print Book2 info
printBook( Book2 );
return 0;
}
void printBook( struct Books book ) {
cout << "Book title : " << book.title << endl;
cout << "Book author : " << book.author << endl;
cout << "Book subject : " << book.subject << endl;
cout << "Book id : " << book.book_id << endl;
}
```

When the above code is compiled and executed, it produces the following result −

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```

Pointers to Structures

You can define pointers to structures in very similar way as you define pointer to any other variable as follows −

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows −

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows −

```
struct_pointer->title;
```

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept −

Example:

```
#include < iostream >
using namespace std;
#include < cstring >
using namespace std;
void printBook( struct Books *book );
struct Books {
char title[50];
char author[50];
char subject[100];
int book_id;
};
int main() {
struct Books Book1; // Declare Book1 of type Book
struct Books Book2; // Declare Book2 of type Book
// Book 1 specification
strcpy( Book1.title, "Learn C++ Programming");
strcpy( Book1.author, "Chand Miyan");
strcpy( Book1.subject, "C++ Programming");
Book1.book_id = 6495407;
// Book 2 specification
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Yakit Singha");
strcpy( Book2.subject, "Telecom");
Book2.book_id = 6495700;
// Print Book1 info, passing address of structure
printBook( &Book1 );
// Print Book1 info, passing address of structure
printBook( &Book2 );
return 0;
}
// This function accept pointer to structure as parameter.
void printBook( struct Books *book ) {
cout << "Book title : " << book->title << endl;
cout << "Book author : " << book->author << endl;
cout << "Book subject : " << book->subject << endl;
cout << "Book id : " << book->book_id << endl;
}
```

When the above code is compiled and executed, it produces the following result −

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```

The typedef Keyword

There is an easier way to define structs or you could "alias" types you create. For example −

```
typedef struct {
char title[50];
char author[50];
char subject[100];
int book_id;
} Books;
```

Now, you can use Books directly to define variables of Books type without using struct keyword. Following is the example −

```
Books Book1, Book2;
```

You can use typedef keyword for non-structs as well as follows −

```
typedef long int *pint32;
pint32 x, y, z;
```

x, y and z are all pointers to long ints.

OOP Concept

C++ Classses and Objects

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.
A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

C++ Class Definitions
When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.
A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations.

Example:

```
class Box {
public:
double length; // Length of a box
double breadth; // Breadth of a box
double height; // Height of a box
};
```

The keyword public determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object.
You can also specify the members of a class as private or protected which we will discuss in a sub-section.

Define C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box:

```
Box Box1; // Declare Box1 of type Box
Box Box2; // Declare Box2 of type Box
```
Both of the objects Box1 and Box2 will have their own copy of data members.

Accessing the Data Members
The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear.
When this code is compiled and executed, it produces the following result-
```
Volume of Box1 : 210
Volume of Box2 : 1560
```
It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

Example:

```
#include < iostream >
using namespace std;
class Box {
public:
double length; // Length of a box
double breadth; // Breadth of a box
double height; // Height of a box
};
int main() {
Box Box1; // Declare Box1 of type Box
Box Box2; // Declare Box2 of type Box
double volume = 0.0; // Store the volume of a box here
// box 1 specification
Box1.height = 5.0;
Box1.length = 6.0;
Box1.breadth = 7.0;
// box 2 specification
Box2.height = 10.0;
Box2.length = 12.0;
Box2.breadth = 13.0;
// volume of box 1
volume = Box1.height * Box1.length * Box1.breadth;
cout << "Volume of Box1 : " << volume << endl; //volume of box 2
volume=Box2.height * Box2.length * Box2.breadth;
cout << "Volume of Box2 : " << volume << endl;
return 0;
```

Classes and Objects in Detail

So far, you have got very basic idea about C++ Classes and Objects. There are further interesting concepts related to C++ Classes and Objects which we will discuss in various sub-sections listed below:
1. Class Member Functions
A member function of a class is a function that has its definition or its prototype within the class definition like any other variable.

2. Class Access Modifiers
A class member can be defined as public, private or protected. By default members would be assumed as private.

3. Constructor & Destructor
A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted.

4. Copy Constructor
The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.

5. Friend Functions
A friend function is permitted full access to private and protected members of a class.

6. Inline Functions With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function.

7. this Pointer Every object has a special pointer this which points to the object itself.

8. Pointer to C++ Classes
A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it.

9. Static Members of a Class

C++ Inheritance

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.
The idea of inheritance implements the is a relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

Base and Derived Classes
A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:
class derived-class: access-specifier base-class
Where access-specifier is one of public, protected, or private, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Example:

```
#include < iostream >
using namespace std;
// Base class
class Shape {
public:
void setWidth(int w) {
width = w;
}
void setHeight(int h) {
height = h;
}
protected:
int width;
int height;
};
// Derived class
class Rectangle: public Shape {
public:
int getArea() {
return (width * height);
}
};
int main(void) {
Rectangle Rect;
Rect.setWidth(5);
Rect.setHeight(7);
// Print the area of the object.
cout << "Total area: " << Rect.getArea() << endl;
return 0;
}
```

Access Control and Inheritance
A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.
We can summarize the different access types according to - who can access them in the following way:
Same class -- public protected private
Derived classes -- public protected
Outside classes -- public

A derived class inherits all base class methods with the following exceptions−
• Constructors, destructors and copy constructors of the base class.
• Overloaded operators of the base class.
• The friend functions of the base class.

Type of Inheritance

When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance. The type of inheritance is specified by the access-specifier as explained above.
We hardly use protected or private inheritance, but public inheritance is commonly used. While using different type of inheritance, following rules are applied-

• Public Inheritance − When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

• Protected Inheritance − When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.

• Private Inheritance − When deriving from a private base class, public and protected members of the base class become private members of the derived class.

Multiple Inheritance

A C++ class can inherit members from more than one class and here is the extended syntax:
class derived-class: access baseA, access baseB....

Example:

```
#include < iostream >
using namespace std;
// Base class Shape
class Shape {
public:
void setWidth(int w) {
width = w;
}
void setHeight(int h) {
height = h;
}
protected:
int width;
int height;
};
// Base class PaintCost
class PaintCost {
public:
int getCost(int area) {
return area * 70;
}
};
// Derived class
class Rectangle: public Shape, public PaintCost {
public:
int getArea() {
return (width * height);
}
};
int main(void) {
Rectangle Rect;
int area;
Rect.setWidth(5);
Rect.setHeight(7);
area = Rect.getArea();
// Print the area of the object.
cout << "Total area: " << Rect.getArea() << endl;
// Print the total cost of painting
cout << "Total paint cost: $" << Rect.getCost(area) << endl;
return 0;
}
```

C++ Overloading(Operator and Function)

C++ allows you to specify more than one definition for a function name or an operator in the same scope, which is called function overloading and operator overloading respectively.
An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).
When you call an overloaded function or operator, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions.
The process of selecting the most appropriate overloaded function or operator is called overload resolution.

Function Overloading in C++
You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.
Following is the example where same function print() is being used to print different data types:

Example:

```
#include < iostream >
using namespace std;
class printData {
public:
void print(int i) {
cout << "Printing int: " << i << endl;
}
void print(double f) {
cout << "Printing float: " << f << endl;
}
void print(char* c) {
cout << "Printing character: " << c << endl;
}
};
```

```
int main(void) {
printData pd;
// Call print to print integer
pd.print(5);
// Call print to print float
pd.print(500.263);
// Call print to print character
pd.print("Hello C++");
return 0;
}
```

Operators Overloading in C++
You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.
Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.
Box operator+(const Box&);
declares the addition operator that can be used to add two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions.
In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows:
Box operator+(const Box&, const Box&);

Example:

```
#include < iostream >
using namespace std;
class Box {
public:
double getVolume(void) {
return length * breadth * height;
}
void setLength( double len ) {
length = len;
}
void setBreadth( double bre ) {
breadth = bre;
}
void setHeight( double hei ) {
height = hei;
}
// Overload + operator to add two Box objects.
Box operator+(const Box& b) {
Box box;
box.length = this->length + b.length;
box.breadth = this->breadth + b.breadth;
box.height = this->height + b.height;
return box;
}
private:
double length; // Length of a box
double breadth; // Breadth of a box
double height; // Height of a box
};
// Main function for the program
int main() {
Box Box1; // Declare Box1 of type Box
Box Box2; // Declare Box2 of type Box
Box Box3; // Declare Box3 of type Box
double volume = 0.0; // Store the volume of a box here
// box 1 specification
Box1.setLength(6.0);
Box1.setBreadth(7.0);
Box1.setHeight(5.0);
// box 2 specification
Box2.setLength(12.0);
Box2.setBreadth(13.0);
Box2.setHeight(10.0);
// volume of box 1
volume = Box1.getVolume();
cout << "Volume of Box1 : " << volume << endl; //volume of box 2
volume=Box2.getVolume();
cout << "Volume of Box2 : " << volume << endl;
//Add two object as follows:
Box3 = Box1 + Box2;
//volume of box 3
volume = Box3.getVolume();
cout << "Volume of Box3 : " << volume << endl;
return 0;
}
```

Buffered Classes

Overloadable/Non-overloadable Operators

Following is the list of operators which can be overloaded:

```
+ - * / % ^
& | ~ ! , =
< > <= >= ++ --
<< >> == != && ||
+= -= /= %= ^= &=
|= *= <<= >>= [] ()
-> ->* new new [] delete delete []
```

Following is the list of operators, which can not be overloaded:
:: .* . ?:

Operator Overloading Examples
Here are various operator overloading examples to help you in understanding the concept.

1. Unary Operators Overloading
2. Binary Operators Overloading
3. Relational Operators Overloading
4. Input/Output Operators Overloading
5. ++ and -- Operators Overloading
6. Assignment Operators Overloading
7. Function call () Operator Overloading
8. Subscripting [] Operator Overloading
9. Class Member Access Operator -> Overloading

C++ Polymorphism

The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.
C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Example:

```
#include < iostream >
using namespace std;
class Shape {
protected:
int width, height;
public:
Shape( int a = 0, int b = 0){
width = a;
height = b;
}
int area() {
cout << "Parent class area :" << endl;
return 0;
}
};
class Rectangle: public Shape {
public:
Rectangle( int a = 0, int b = 0):Shape(a, b) { }
int area () {
cout << "Rectangle class area :" << endl;
return (width * height);
}
};
class Triangle: public Shape {
public:
Triangle( int a = 0, int b = 0):Shape(a, b) { }
int area () {
cout << "Triangle class area :" << endl;
return (width * height / 2);
}
};
// Main function for the program
int main() {
Shape *shape;
Rectangle rec(10,7);
Triangle tri(10,5);
// store the address of Rectangle
shape = &rec;
// call rectangle area.
shape->area();
// store the address of Triangle
shape = &tri;
// call triangle area.
shape->area();
return 0;
}
```

When the above code is compiled and executed, it produces the following result-

```
Parent class area :
Parent class area :
```

The reason for the incorrect output is that the call of the function area() is

being set once by the compiler as the version defined in the base class. This is called static resolution of the function call, or static linkage - the function call is fixed before the program is executed.
This is also sometimes called early binding because the area() function is set during the compilation of the program.
But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword virtual so that it looks like this:

```
class Shape {
protected:
int width, height;
public:
Shape( int a = 0, int b = 0) {
width = a;
height = b;
}
virtual int area() {
cout << "Parent class area :" << endl;
return 0;
}
};
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result:

```
Rectangle class area
Triangle class area
```

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.
As you can see, each of the child classes has a separate implementation for the function area(). This is how polymorphism is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

StrVirtual Functioneams

A virtual function is a function in a base class that is declared using the keyword virtual. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.
What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called.

This sort of operation is referred to as dynamic linkage, or late binding.

Pure Virtual Functions
It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.
We can change the virtual function area() in the base class to the following:

```
class Shape {
protected:
int width, height;
public:
Shape(int a = 0, int b = 0) {
width = a;
height = b;
}
// pure virtual function
virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called pure virtual function.

Data Abstraction in C++

Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.
Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.
Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.
Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having any knowledge of its internals.
In C++, classes provides great level of data abstraction. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

For example, your program can make a call to the sort() function without

knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.
In C++, we use classes to define our own abstract data types (ADT). You can use the cout object of class ostream to stream data to standard output like this:

```
#include < iostream >
using namespace std;
int main() {
cout << "Hello C++" << endl;
return 0;
}
```

Here, you don't need to understand how cout displays the text on the user's screen. You need to only know the public interface and the underlying implementation of ‘cout’ is free to change.

Access Labels Enforce Abstraction
In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels:
• Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
• Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.
There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Data abstraction provides two important advantages −
• Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
• The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.
By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data is public, then any function that directly access the data members of the old representation might be broken.

Example:

```
#include < iostream>
using namespace std;
class Adder {
public:
// constructor
Adder(int i = 0) {
total = i;
}
// interface to outside world
void addNum(int number) {
total += number;
}
// interface to outside world
int getTotal() {
return total;
};
private:
// hidden data from outside world
int total;
};
int main() {
Adder a;
a.addNum(10);
a.addNum(20);
a.addNum(30);
cout << "Total " << a.getTotal() << endl;
return 0;
}
```

Designing Strategy
Abstraction separates code into interface and implementation. So while designing your component, you must keep interface independent of the implementation so that if you change underlying implementation then interface would remain intact.
In this case whatever programs are using these interfaces, they would not be impacted and would just need a recompilation with the latest implementation.

Data Encapsulation in C++

All C++ programs are composed of the following two fundamental elements:

• Program statements (code) − This is the part of a program that performs actions and they are called functions.
• Program data − The data is the information of the program which gets affected by the program functions.
Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding.
Data encapsulation is a mechanism of bundling the data, and the functions that use them and data abstraction is a mechanism of exposing only the interfaces and hiding the implementation details from the user.
C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes. We already have studied that a class can contain private, protected and public members. By default, all items defined in a class are private.
For example:

```
class Box {
public:
double getVolume(void) {
return length * breadth * height;
}
private:
double length; // Length of a box
double breadth; // Breadth of a box
double height; // Height of a box
};
```

The variables length, breadth, and height are private. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.
To make parts of a class public (i.e., accessible to other parts of your program), you must declare them after the public keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.
Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.

Example:

```
#include < iostream >
using namespace std;
class Adder {
public:
```

```
// constructor
Adder(int i = 0) {
total = i;
}
// interface to outside world
void addNum(int number) {
total += number;
}
// interface to outside world
int getTotal() {
return total;
};
private:
// hidden data from outside world
int total;
};
int main() {
Adder a;
a.addNum(10);
a.addNum(20);
a.addNum(30);
cout << "Total " << a.getTotal() << endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Total 60
```

Above class adds numbers together, and returns the sum. The public members addNum and getTotal are the interfaces to the outside world and a user needs to know them to use the class.
The private member total is something that is hidden from the outside world, but is needed for the class to operate properly.

Designing Strategy:
Most of us have learnt to make class members private by default unless we really need to expose them. That's just good encapsulation.
This is applied most frequently to data members, but it applies equally to all members, including virtual functions.

Interface in C++

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.
The C++ interfaces are implemented using abstract classes and these

abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.
A class is made abstract by declaring at least one of its functions as pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration as follows −

```
class Box {
public:
// pure virtual function
virtual double getVolume() = 0;
private:
double length; // Length of a box
double breadth; // Breadth of a box
double height; // Height of a box
};
```

The purpose of an abstract class (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an interface. Attempting to instantiate an object of an abstract class causes a compilation error.
Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC. Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.
Classes that can be used to instantiate objects are called concrete classes.

Example:

```
#include < iostream >
using namespace std;
// Base class
class Shape {
public:
// pure virtual function providing interface framework.
virtual int getArea() = 0;
void setWidth(int w) {
width = w;
}
void setHeight(int h) {
height = h;
}
protected:
int width;
int height;
};
```

```
// Derived classes
class Rectangle: public Shape {
public:
int getArea() {
return (width * height);
}
};
class Triangle: public Shape {
public:
int getArea() {
return (width * height)/2;
}
};
int main(void) {
Rectangle Rect;
Triangle Tri;
Rect.setWidth(5);
Rect.setHeight(7);
// Print the area of the object.
cout << "Total Rectangle area: " << Rect.getArea() << endl;
Tri.setWidth(5);
Tri.setHeight(7);
// Print the area of the object.
cout << "Total Triangle area: " << Tri.getArea() << endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Total Rectangle area: 35
Total Triangle area: 17
```

You can see how an abstract class defined an interface in terms of getArea() and two other classes implemented same function but with different algorithm to calculate the area specific to the shape.

Designing Strategy:

An object-oriented system might use an abstract base class to provide a common and standardized interface appropriate for all the external applications. Then, through inheritance from that abstract base class, derived classes are formed that operate similarly.

The capabilities (i.e., the public functions) offered by the external applications are provided as pure virtual functions in the abstract base class. The implementations of these pure virtual functions are provided in the derived classes that correspond to the specific types of the application.