Unit I: Introduction: OOP concept, Procedural vs OOP programming, OOP terminology and features(data encapsulation, inheritance, polymorphism and late binding), Tokens, Character set, Keywords, Data-types, Data Types declarations, Constants and variables, expressions, Standard Library and header files. Objects & Classes in C++: Declaring & using classes, Constructors, Objects as functions arguments, Copy Constructor, Static class data. Arrays of objects, C++ String class.

What does Object-Oriented Programming Language (OOPL) mean?

Object-oriented programming language (OOPL) is a high-level programming language based on the object-oriented programming (OOP) model.

OOPL incorporates logical classes, objects, methods, relationships and other processes with the design of software and applications. The first OOPL was Simula, a simulation creation tool developed in 1960.

Benefits of C++ over C Language

The major difference being OOPS concept, C++ is an object oriented language whereas C language is a procedural language. Apart from this there are many other features of C++ which gives this language an upper hand on C language.

Following features of C++ makes it a stronger language than C,

1. There is Stronger Type Checking in C++.
2. All the OOPS features in C++ like Abstraction, Encapsulation, Inheritance etc makes it more worthy and useful for programmers.
3. C++ supports and allows user defined operators (i.e. Operator Overloading) and function overloading is also supported in it.
4. Exception Handling is there in C++.
5. The Concept of Virtual functions and also Constructors and Destructors for Objects.
6. Inline Functions in C++ instead of Macros in C language. Inline functions make complete function body act like Macro, safely.
7. Variables can be declared anywhere in the program in C++, but must be declared before they are used.

OOPS Concept

Some of the main features of Object Oriented Programming which you will be using in C++.

1. Objects
2. Classes
3. Abstraction
4. Encapsulation
5. Inheritance
6. Overloading
7. Exception Handlin

Objects

Objects are the basic unit of OOP. They are instances of class, which have data members and uses various member functions to perform tasks.

Class

It is similar to structures in C language. Class can also be defined as user defined data type but it also contains functions in it. So, class is basically a blueprint for object. It declares & defines what data variables the object will have and what operations can be performed on the class's object.

Abstraction

Abstraction refers to showing only the essential features of the application and hiding the details. In C++, classes provide methods to the outside world to access & use the data variables, but the variables are hidden from direct access. This can be done access specifiers.

Encapsulation

It can also be said data binding. Encapsulation is all about binding the data variables and functions together in class.

Inheritance

Inheritance is a way to reuse once written code again and again. The class which is inherited is called base calls & the class which inherits is called derived class. So when, a derived class inherits a base class, the derived class can use all the functions which are defined in base class, hence making code reusable.

Polymorphism

It is a feature, which lets us create functions with same name but different arguments, which will perform differently. That is function with same name, functioning in different way. Or, it also allows us to redefine a function to provide its new definition. You will learn how to do this in details soon in coming lessons.

Exception Handling

Exception handling is a feature of OOP, to handle unresolved exceptions or errors produced at runtime.

Difference Between Procedure Oriented Programming (POP) & Object Oriented Programming (OOP)

	Procedure Oriented Programming	**Object Oriented Programming**
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP, Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Examples of POP are: C, VB, FORTRAN, and Pascal.	Examples of OOP are: C++, JAVA, VB.NET, C#.NET.

Data Types in C++

They are used to define type of variables and contents used. Data types define the way you use storage in the programs you write. Data types can be built in or abstract.

Built in Data Types

These are the data types which are predefined and are wired directly into the compiler. eg: int, char etc.

User defined or Abstract data types

These are the type, that user creates as a class. In C++ these are classes where as in C it was implemented by structures.

Basic Built in types

char	for character storage (1 byte)
int	for integral number (2 bytes)
float	single precision floating point (4 bytes)
double	double precision floating point numbers (8 bytes)

Enum as Data type

Enumerated type declares a new type-name and a sequence of value containing identifiers which has values starting from 0 and incrementing by 1 every time.

For Example :

enum day(mon, tues, wed, thurs, fri) d;

Here an enumeration of days is defined with variable d. *mon* will hold value 0, *tue* will have 1 and so on. We can also explicitly assign values, like, enum day(mon, tue=7, wed);. Here, *mon* will be 0, *tue* is assigned 7, so *wed* will have value 8.

Basic types of Variables

Each variable while declaration must be given a datatype, on which the memory assigned to the variable depends. Following are the basic types of variables,

`bool`	For variable to store boolean values(True or False)
`char`	For variables to store character types.
`int`	for variable with integral values

`float` and `double` are also types for variables with large and floating point values

Scope of Variables

All the variables have their area of functioning, and out of that boundary they don't hold their value, this boundary is called scope of the variable. For most of the cases it’s between the curly braces, in which variable is declared that a variable exists, not outside it. We can broadly divide variables into two main types,

- Global Variables
- Local variables

Global variables

Global variables are those, which ar once declared and can be used throughout the lifetime of the program by any class or any function. They must be declared outside the main () function. If only declared, they can be assigned different values at different time in program lifetime. But even if they are declared and initialized at the same time outside the main () function, then also they can be assigned any value at any point in the program.

Local Variables

Local variables are the variables which exist only between the curly braces, in which it's declared. Outside that they are unavailable and lead to compile time error.

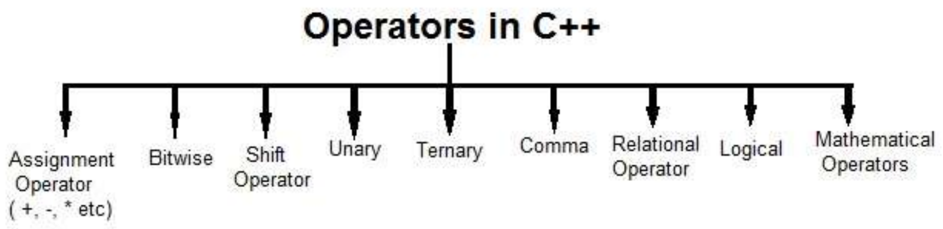
Some special types of variable

There are also some special keywords, to impart unique characteristics to the variables in the program. Following two are mostly used; we will discuss them in details later.

1. **Final** - Once initialized, its value can't be changed.
2. **Static** - These variables holds their value between function calls.

Operators in C++

Operators are special type of functions that takes one or more arguments and produces a new value. For example: addition (+), subtraction (-), multiplication (*) etc, are all operators. Operators are used to perform various operations on variables and constants.



Types of operators

1. Assignment Operator
2. Mathematical Operators
3. Relational Operators
4. Logical Operators
5. Bitwise Operators
6. Shift Operators
7. Unary Operators
8. Ternary Operator
9. Comma Operator

C++ Basic Elements

Programming language is a set of rules, symbols, and special words used to construct programs. There are certain elements that are common to all programming languages.

C++ Character Set

Character set is a set of valid characters that a language can recognize.

Letters	A-Z, a-z
Digits	0-9
Special Characters	Space + - * / ^ \ () [] {} = != <> ‘ “ $, ; : % ! & ? _ # <= >= @
Formatting characters	backspace, horizontal tab, vertical tab, form feed, and carriage return

Tokens

A token is a group of characters that logically belong together. The programmer can write a program by using tokens. C++ uses the following types of tokens. Keywords, Identifiers, Literals, Punctuators, Operators.

1. Keywords

These are some reserved words in C++ which have predefined meaning to compiler called keywords. It is discussed in previous section.

2. Identifiers

Symbolic names can be used in C++ for various data items used by a programmer in his program. A symbolic name is generally known as an identifier. The identifier is a sequence of characters taken from C++ character set. The rules for the formation of an identifier are:

- An identifier can consist of alphabets, digits and/or underscores.
- It must not start with a digit
- C++ is case sensitive that is upper case and lower case letters are considered different from each other.
- It should not be a reserved word.

3. Literals

Literals (often referred to as constants) are data items that never change their value during the execution of the program. The following types of literals are available in C++.

- Integer-Constants
- Character-constants
- Floating-constants
- Strings-constants

Integer Constants

Integer constants are whole number without any fractional part. C++ allows three types of integer constants.

Decimal integer constants : It consists of sequence of digits and should not begin with 0 (zero). For example 124, - 179, +108.

Octal integer constants: It consists of sequence of digits starting with 0 (zero). For example. 014, 012.

Hexadecimal integer constant: It consists of sequence of digits preceded by ox or OX.

Character constants

A character constant in C++ must contain one or more characters and must be enclosed in single quotation marks. For example 'A', '9', etc. C++ allows no graphic characters which cannot be typed directly from keyboard, e.g., backspace, tab, carriage return etc. These characters can be represented by using an escape sequence. An escape sequence represents a single character.

Floating constants

They are also called real constants. They are numbers having fractional parts. They may be written in fractional form or exponent form. A real constant in fractional form consists of signed or unsigned digits including a decimal point between digits. For example 3.0, -17.0, -0.627 etc.

String Literals

A sequence of character enclosed within double quotes is called a string literal. String literal is by default (automatically) added with a special character ‘\0' which denotes the end of the string. Therefore the size of the string is increased by one character. For example "COMPUTER" will re represented as "COMPUTER\0" in the memory and its size is 9 characters.

4. Punctuators

The following characters are used as punctuators in C++.

Brackets [ ]	Opening and closing brackets indicate single and multidimensional array subscript.
Parentheses ( )	Opening and closing brackets indicate functions calls,; function parameters for grouping expressions etc.
Braces { }	Opening and closing braces indicate the start and end of a compound statement.
Comma ,	It is used as a separator in a function argument list.
Semicolon ;	It is used as a statement terminator.
Colon :	It indicates a labeled statement or conditional operator symbol.
Asterisk *	It is used in pointer declaration or as multiplication operator.
Equal sign =	It is used as an assignment operator.
Pound sign #	It is used as pre-processor directive.

Introduction to Classes and Objects

The classes are the most important feature of C++ that leads to Object Oriented programming. Class is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating instance of that class.

The variables inside class definition are called as data members and the functions are called member functions.

More about Classes

1. Class name must start with an uppercase letter. If class name is made of more than one word, then first letter of each word must be in uppercase. *Example,*

   Class Study, class StudyTonight etc

2. Classes contain data members and member functions, and the access of these data members and variable depends on the access specifiers (discussed in next section).

3. Class's member functions can be defined inside the class definition or outside the class definition.

4. Classes in C++ are similar to structures in C, the only difference being, class defaults to private access control, where as structure defaults to public.
5. All the features of OOPS, revolve around classes in C++. Inheritance, Encapsulation, Abstraction etc.
6. Objects of class holds separate copies of data members. We can create as many objects of a class as we need.
7. Classes do posses more characteristics, like we can create abstract classes, immutable classes,

Objects

Class is mere a blueprint or a template. No storage is assigned when we define a class. Objects are instances of class, which holds the data variables declared in class and the member functions work on these class objects.

Each object has different data variables. Objects are initialised using special class functions called **Constructors**. We will study about constructors later.

And whenever the object is out of its scope, another special class member function called **Destructor** is called, to release the memory reserved by the object.

Access Control in Classes

Now before studying how to define class and its objects, lets first quickly learn what are access specifiers.

Access specifier in C++ class defines the access control rules. C++ has 3 new keywords introduced, namely,

1. public
2. private
3. protected

These access specifiers are used to set boundaries for availability of members of class be it data members or member functions

Access specifiers in the program, are followed by a colon. You can use one, two or all 3 specifiers in the same class to set different boundaries for different class members. They change the boundary for all the declarations that follow them.

Public

Public, means all the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. Hence there are chances that they might change them. So the key members must not be declared public.

```
class PublicAccess

{

        public:   // public access specifier
        int x;           // Data Member Declaration
        void display();   // Member Function decaration
}
```

Private

Private keyword, means that no one can access the class members declared private outside that class. If someone tries to access the private member, they will get a compile time error. By default class variables and member functions are private.

```
class PrivateAccess

{

        private:   // private access specifier
        int x;           // Data Member Declaration
        void display();   // Member Function decaration
}
```

Protected

Protected, is the last access specifier, and it is similar to private, it makes class member inaccessible outside the class. But they can be accessed by any subclass of that class. (If class A is inherited by class B, then class B is subclass of class A. We will learn this later.)

```
class ProtectedAccess

{

        protected:   // protected access specifier
        int x;           // Data Member Declaration
        void display();   // Member Function decaration
}
```

Constructors

Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created. Constructors iitialize values to object members after storage is allocated to the object.

```
class A
{
        int x;
        public:
        A();  //Constructor
};
```

While defining a contructor you must remeber that the name of constructor will be same as the name of the class, and contructors never have return type.

Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator.

```
class A

{

        int i;
        public:
        A(); //Constructor declared
};

A::A()   // Constructor definition
{
        i=1;
}
```

Types of Constructors

Constructors are of three types:

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor

Default Constructor

Default constructor is the constructor which doesn't take any argument. It has no parameter

Parameterized Constructor

These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.

Copy Constructor

These are special type of Constructors which takes an object as argument, and is used to copy values of data members of one object into other object.

Copy Constructor in C++

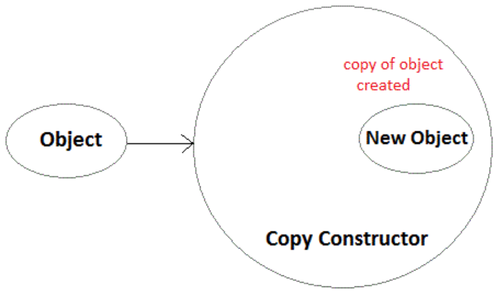
Copy Constructor is a type of constructor which is used to create a copy of an already existing object of a class type. It is usually of the form X (X&), where X is the class name. The compiler provides a default Copy Constructor to all the classes.

Syntax of Copy Constructor

Class-name (*class-name &*)

```
{
        . . . .
}
```

As it is used to create an object, hence it is called a constructor. And, it creates a new object, which is exact copy of the existing copy, hence it is called **copy constructor**.



Destructors

Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope.

The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor; with a **tilde** ~ sign as prefix to it.

```
class A
{
        public:
        ~A();
};
```

Destructors will never have any arguments.

Static Keyword

Static is a keyword in C++ used to give special characteristics to an element. Static elements are allocated storage only once in a program lifetime in static storage area. And they have a scope till the program lifetime. Static Keyword can be used with following,

1. Static variable in functions
2. Static Class Objects
3. Static member Variable in class
4. Static Methods in class

Static variables inside Functions

Static variables when used inside function are initialized only once, and then they hold their value even through function calls.

These static variables are stored on static storage area, not in stack.

Static class Objects

Static keyword works in the same way for class objects too. Objects declared static are allocated storage in static storage area, and have scope till the end of program.

Static objects are also initialized using constructors like other normal objects. Assignment to zero, on using static keyword is only for primitive datatypes, not for user defined datatypes.

Static data member in class

Static data members of class are those members which are shared by all the objects. Static data member has a single piece of storage, and is not available as separate copy with each object, like other non-static data members.

Static member variables (data members) are not initialized using constructor, because these are not dependent on object initialization.

Also, it must be initialized explicitly, always outside the class. If not initialized, Linker will give error.

Static Member Functions

These functions work for the class as whole rather than for a particular object of a class.

It can be called using an object and the direct member access . operator. But, it’s more typical to call a static member function by itself, using class name and scope resolution :: operator. These functions cannot access ordinary data members and member functions, but only static data members and static member functions. It doesn't have any "this" keyword which is the reason it cannot access ordinary members.

Objects as Function Arguments in c++

The objects of a class can be passed as arguments to member functions as well as nonmember functions either by value or by reference. When an object is passed by value, a copy of the actual object is created inside the function. This copy is destroyed when the function terminates. Moreover, any changes made to the copy of the object inside the function are not reflected in the actual object. On the other hand, in pass by reference, only a reference to that object (not the entire object) is passed to the function. Thus, the changes made to the object within the function are also reflected in the actual object.

Whenever an object of a class is passed to a member function of the same class, its data members can be accessed inside the function using the object name and the dot operator. However, the data members of the calling object can be directly accessed inside the function without using the object name and the dot operator.

To understand how objects are passed and accessed within a member function, consider this example.

```
#include <iostream.h>
class rational
{
        private:
        int num;
        int dnum;

        public:
        rational():num(1),dnum(1)
        {}
        void get ()
        {
                cout<<"enter numerator";
                cin>>num;
                cout<<"enter denomenator";
                cin>>dnum;
        }
        void print ()
        {
                cout<<num<<"/"<<dnum<<endl;
        }
        void multi(rational r1,rational r2)
        {
                num=r1.num*r2.num;
                dnum=r1.dnum*r2.dnum;
        }
};
void main ()
{
        rational r1,r2,r3;
        r1.get();
        r2.get();
        r3.multi(r1,r2);
        r3.print();

}
```

Array of Objects in c++

Like array of other user-defined data types, an array of type class can also be created. The array of type class contains the objects of the class as its individual elements. Thus, an array of a class type is also known as an array of objects. An array of objects is declared in the same way as an array of any built-in data type.

The syntax for declaring an array of objects is

class_name array_name [size];

To understand the concept of an array of objects, consider this example.

```
#include<iostream>
using namespace std;
class books
{
        char tit1e [30];
        float price ;
        public:
        void getdata ();
        void putdata ();
} ;
void books :: getdata ()
{
        cout<<"Title:”;
        Cin>>title;
        cout<<"Price:”;
        cin>>price;

}
void books :: putdata ()
{
        cout<<"Title:"<<title<< "\n";
        cout<<"Price:"<<price<< "\n”;
        const int size=3 ;
}

int main ()
{
books book[size] ;
for(int i=0;i<size;i++)
{
        cout<<"Enter details o£ book "<<(i+1)<<"\n";
        book[i].getdata();
}
for(int i=0;i<size;i++)
{
        cout<<"\nBook "<<(i+l)<<"\n";
        book[i].putdata() ;
}
return 0;
}
```

Unit II: Operator overloading: Overloading unary & binary operators. Data conversion. Pitfalls of operator overloading. Pointers & arrays. Pointers & functions. New& delete operators. Pointers for objects.

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 had been 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 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.

```
#include <iostream.h>

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.

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 cannot be overloaded:

::	.*	.	?:

```
#include <iostream.h>

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;
}
```

Unary operators overloading in C++

The unary operators operate on a single operand and following are the examples of Unary operators:

- The increment (++) and decrement (--) operators.
- The unary minus (-) operator.
- The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

```
#include <iostream.h>

class Distance
{
   private:
      int feet;             // 0 to infinite
      int inches;
        // 0 to 12
   public:
      // required constructors
      Distance()
      {
         feet = 0;
         inches = 0;
      }

      Distance(int f, int i)
      {
         feet = f;
         inches = i;
      }

      // method to display distance
      void displayDistance()
      {
         cout << "F: " << feet << " I:" << inches <<endl;
      }
      // overloaded minus (-) operator
      Distance operator- ()
      {
         feet = -feet;
         inches = -inches;
         return Distance(feet, inches);
      }
};

int main()
{
   Distance D1 (11, 10), D2 (-5, 11);

   -D1;                      // apply negation
   D1.displayDistance();     // display D1
   -D2;                     // apply negation
   D2.displayDistance();    // display D2

   return 0;
}
```

Binary operators overloading in C++

The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.
Following example explains how addition (+) operator can be overloaded. Similar way, you can overload subtraction (-) and division (/) operators.

```
#include <iostream.h>

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;
}
```

Inheritance in C++

Inheritance is the capability of one class to acquire properties and characteristics from another class. The class whose properties are inherited by other class is called the **Parent** or **Base** or **Super** class. And, the class which inherits properties of other class is called **Child** or **Derived** or **Sub** class.

Inheritance makes the code reusable. When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused.

Purpose of Inheritance

1. Code Reusability
2. Method Overriding (Hence, Runtime Polymorphism.)
3. Use of Virtual Keyword

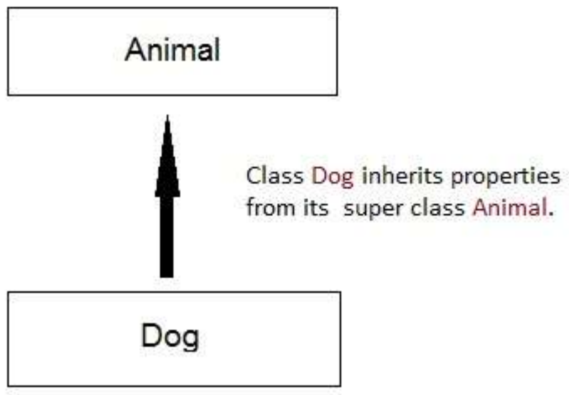
Basic Syntax of Inheritance

class Subclass_name : access_mode Superclass_name

While defining a subclass like this, the super class must be already defined or atleast declared before the subclass declaration.

Access Mode is used to specify, the mode in which the properties of superclass will be inherited into subclass, public, privtate or protected.

Example of Inheritance



Inheritance Visibility Mode

Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public.

1) Public Inheritance

This is the most used inheritance mode. In this the protected member of super class becomes protected members of sub class and public becomes public.

class Subclass : **public** Superclass

2) Private Inheritance

In private mode, the protected and public members of super class become private members of derived class.

class Subclass : Superclass   // By default its private inheritance

3) Protected Inheritance

In protected mode, the public and protected members of Super class becomes protected members of Sub class.

class subclass : **protected** Superclass

Table showing all the Visibility Modes

	Derived Class	**Derived Class**	**Derived Class**
Base class	**Public Mode**	**Private Mode**	**Protected Mode**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected

	Derived Class	**Derived Class**	**Derived Class**
Base class	**Public Mode**	**Private Mode**	**Protected Mode**
Public	Public	Private	Protected

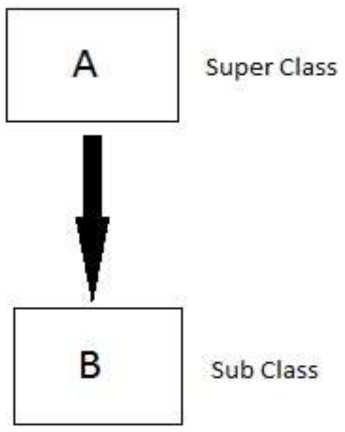
Types of Inheritance

In C++, we have 5 different types of Inheritance. Namely,

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

Single Inheritance

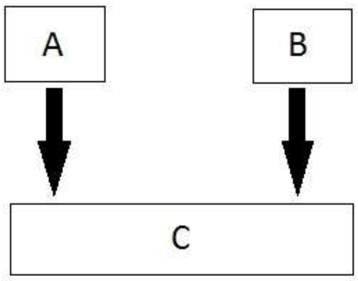
In this type of inheritance one derived class inherits from only one base class. It is the simplest form of Inheritance.



```
#include <iostream.h>
// 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;
}
```

Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes.



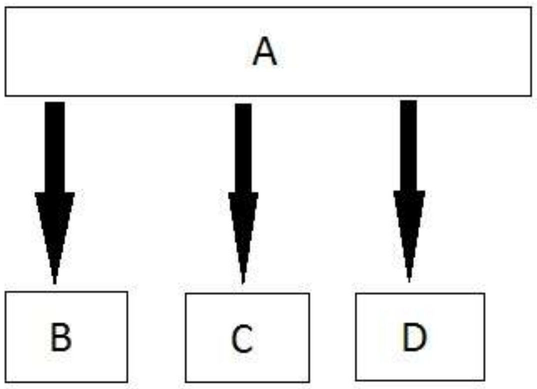
```
#include<iostream.h>
#include<conio.h>

class student
{
    protected:
       int rno,m1,m2;
    public:
                void get()
              {
                    cout<<"Enter the Roll no :";
                    cin>>rno;
                    cout<<"Enter the two marks   :";
                    cin>>m1>>m2;
          }
};
class sports
{
    protected:
      int sm;                  // sm = Sports mark
    public:
            void getsm()
          {
            cout<<"\nEnter the sports mark :";
            cin>>sm;

          }
};
class statement:public student,public sports
{
    int tot,avg;
    public:
    void display()
          {
            tot=(m1+m2+sm);
            avg=tot/3;
            cout<<"\n\n\tRoll No    : "<<rno<<"\n\tTotal      : "<<tot;
           cout<<"\n\tAverage    : "<<avg;
          }
};
void main()
{
   clrscr();
   statement obj;
   obj.get();
   obj.getsm();
   obj.display();
   getch();
}
```

Output:

```
          Enter the Roll no: 100

          Enter two marks

          90
          80

          Enter the Sports Mark: 90

          Roll No: 100
          Total    : 260
          Average: 86.66
```

Hierarchical Inheritance

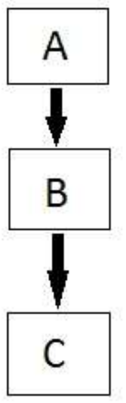
In this type of inheritance, multiple derived classes inherits from a single base class.



```
#include <iostream.h>

#include <conio.h>

class person

{

    char name[100],gender[10];

    int age;

    public:

        void getdata()

        {

            cout<<"Name: ";

            gets(name);

            cout<<"Age: ";

            cin>>age;

            cout<<"Gender: ";

            cin>>gender;

        }

        void display()

        {
            cout<<"Name: "<<name<<endl;
            cout<<"Age: "<<age<<endl;
            cout<<"Gender: "<<gender<<endl;
        }
};

class student: public person
{
    char institute[100], level[20];
    public:
        void getdata()
        {
            person::getdata();
            cout<<"Name of College/School: ";
            gets(institute);
            cout<<"Level: ";
            cin>>level;
        }
        void display()
        {
            person::display();
            cout<<"Name of College/School: "<<institute<<endl;
            cout<<"Level: "<<level<<endl;
        }
};

class employee: public person
{
    char company[100];
    float salary;
    public:
        void getdata()
        {
            person::getdata();
            cout<<"Name of Company: ";
            gets(company);
            cout<<"Salary: Rs.";
            cin>>salary;
        }
        void display()
        {
            person::display();
            cout<<"Name of Company: "<<company<<endl;
            cout<<"Salary: Rs."<<salary<<endl;
        }
};

int main()
{
    student s;
    employee e;
    cout<<"Student"<<endl;
    cout<<"Enter data"<<endl;
    s.getdata();
    cout<<endl<<"Displaying data"<<endl;
    s.display();
    cout<<endl<<"Employee"<<endl;
    cout<<"Enter data"<<endl;
    e.getdata();
    cout<<endl<<"Displaying data"<<endl;
    e.display();

    getch();

    return 0;

}
```

Multilevel Inheritance

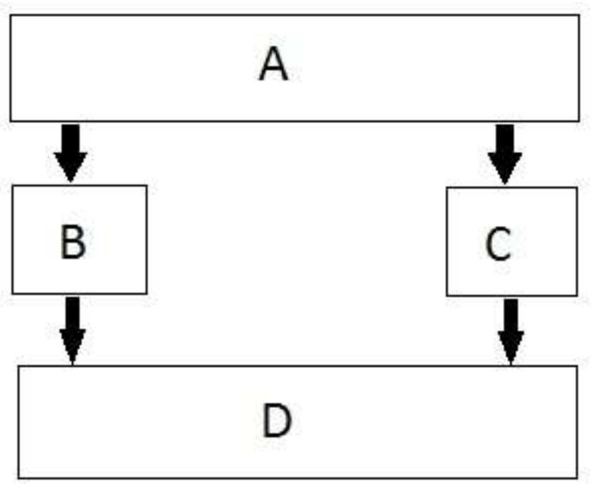
In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



```
#include<iostream>
#include<conio.h>
class Father
{
        protected:
                int a;
        public:
                Father()
                {
                        a = 5;
        }
};

class Son : public Father
{
        protected:
                int b;
        public:
                Son() : Father()
                {
                        b = 9;
                }
};

class GrandSon : public Son
{
        protected:
                int c;
        public:
                GrandSon() :Son()
                {
                        c = 0;
                        c = a + b;
                }
                void Show()
                {
                        std::cout << " a + b = " << c<<"\n"<<"\n"<<"\n";
                }
};

void main()
{
        GrandSon obj;
        obj.Show();
        getch();
}
```

Hybrid (Virtual) Inheritance

Hybrid Inheritance is combination of Hierarchical and Multilevel Inheritance.



Order of Constructor Call

Base class constructors are always called in the derived class constructors. Whenever you create derived class object, first the base class default constructor is executed and then the derived class's constructor finishes execution.

\

Points to Remember

1. Whether derived class's default constructor is called or parameterised is called, base class's default constructor is always called inside them.

2. To call base class's parameterised constructor inside derived class's parameterised constructo, we must mention it explicitly while declaring derived class's parameterized constructor.

Base class Default Constructor in Derived class Constructors

```
class Base
{ int x;
  public:
  Base() { cout << "Base default constructor"; }
};

class Derived : public Base
{ int y;
  public:
  Derived() { cout << "Derived default constructor"; }
  Derived(int i) { cout << "Derived parameterized constructor"; }
};

int main()
{
 Base b;
 Derived d1;
 Derived d2(10);
}
```

In the above example that with both the object creation of the Derived class, Base class's default constructor is called.

Base class Parameterized Constructor in Derived class Constructor

We can explicitly mention to call the Base class's parameterized constructor when Derived class's parameterized constructor is called.

```
class Base
{
        int x;
         public:
         Base(int i)
        {
                x = i;
                cout << "Base Parameterized Constructor";
         }
};

class Derived : public Base
{
        int y;
        public:
        Derived(int j) : Base(j)
        {
        y = j;
         cout << "Derived Parameterized Constructor";
        }
};

int main()
{
        Derived d(10) ;
         cout << d.x ;    // Output will be 10
        cout << d.y ;    // Output will be 10
```

}

Why is Base class Constructor called inside Derived class?

Constructors have a special job of initializing the object properly. A Derived class constructor has access only to its own class members, but a Derived class object also have inherited property of Base class, and only base class constructor can properly initialize base class members. Hence all the constructors are called, else object wouldn't be constructed properly.

Constructor call in Multiple Inheritance

Its almost the same, all the Base class's constructors are called inside derived class's constructor, in the same order in which they are inherited.

class A : public B, public C ;

In this case, first class B constructor will be executed, then class C constructor and then class A constructor.

Hybrid Inheritance and Constructor call

As we all know that whenever a derived class object is instantiated, the base class constructor is always called. But in case of Hybrid Inheritance, as discussed in above example, if we create an instance of class D, then following constructors will be called :

- before class D's constructor, constructors of its super classes will be called, hence constructors of class B, class C and class A will be called.
- when constructors of class B and class C are called, they will again make a call to their super class's constructor.

This will result in multiple calls to the constructor of class A, which is undesirable. As there is a single instance of virtual base class which is shared by multiple classes that inherit from it, hence the constructor of the base class is only called once by the constructor of concrete class, which in our case is class D.

If there is any call for initializing the constructor of class A in class B or class C, while creating object of class D, all such calls will be skipped.

Polymorphism

Polymorphism means having multiple forms of one thing. In inheritance, polymorphism is done, by method overriding, when both super and sub class have member function with same declaration bu different definition.

Function Overriding

If we inherit a class into the derived class and provide a definition for one of the base class's function again inside the derived class, then that function is said to be **overridden**, and this mechanism is called **Function Overriding**

Requirements for Overriding

1. Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.
2. Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

Function Call Binding with class Objects

Connecting the function call to the function body is called **Binding**. When it is done before the program is run, its called **Early** Binding or **Static** Binding or **Compile-time** Binding.

```
class Base
{
             public:
             void shaow()
            {
                         cout << "Base class\t";
            }
};
class Derived:public Base
{
            public:
             void show()
             {
              cout << "Derived Class";
             }
}

int main()
{
            Base b;       //Base class object
            Derived d;     //Derived class object
            b.show();     //Early Binding Ocuurs
             d.show();
}
```

Containership in C++

When a class contains objects of another class or its members, this kind of relationship is called containership or nesting and the class which contains objects of another class as its members is called as container class.

Syntax for the declaration of another class is:

```
Class class_name1
{
        _____
        _____
};
Class class_name2
{
        _____
        ______
};
Class class_name3
{
        Class_name1 obj1;                  // object of class_name1
        Class_name2 obj2;                // object of class_name2
        ______-
        _______
};
```

- In Inheritance, if a class B is derived from class A, B has all characteristics of A in addition to its own.
- Hence you can say that "B is a kind of A". So, inheritance is sometimes called as a "kind of" relationship.
- There is another kind of relationship called a "has a" relationship, or containership. In OOP, this relation occurs when you have an object of class A in class B as shown.

```
//Sample Program to demonstrate Containership

#include < iostream.h >

#include < conio.h >

#include < iomanip.h >

#include< stdio.h >

const int len=80;

class employee

{

        private:

        char name[len];

        int number;

        public:

        void get_data()

        {

                cout << "\n Enter employee name: ";

                cin >> name;

                cout << "\n Enter employee number: ";

                cin >> number;

        }

        void put_data()

        {

                cout << " \n\n Employee name: " << name;

                cout << " \n\n Employee number: " << number;

        }
```

```
};
class manager
{
        private:
        char dept[len];
        int numemp;
        employee emp;
        public:
        void get_data()
        {
                emp.get_data();
                cout << " \n Enter department: ";
                cin >> dept;
                cout << "\n Enter number of employees: ";
                cin >> numemp;
        }
        void put_data()
        {
                emp.put_data();
                cout << " \n\n Department: " << dept;
                cout << " \n\n Number of employees: " << numemp;
        }
};
class scientist
{
        private:
        int pubs,year;
        employee emp;
        public:
        void get_data()
        {
                emp.get_data();
                cout << " \n Number of publications: ";
                cin >> pubs;
                cout << " \n Year of publication: ";
                cin >> year;
        }
        void put_data()
        {
                emp.put_data();
                cout << "\n\n Number of publications: " << pubs;
                cout << "\n\n Year of publication: "<< year;
        }
};

void main()
{
        manager m1;
        scientist s1;
        int ch;
        clrscr();
        do
        {
                cout << "\n 1.manager\n 2.scientist\n";
                cout << "\n   Enter your choice: ";
                cin >> ch;
        switch(ch)
        {
        case 1:
             cout << "\n Manager data:\n";
```

```
                m1.get_data();

                cout << "\n Manager data:\n";

                m1.put_data();

                break;

        case 2:cout << " \n Scientist data:\n";

                s1.get_data();

                cout << " \n Scientist data:\n";

                s1.put_data();

                break;

        }

        cout << "\n\n To continue Press 1 -> ";

        cin >> ch;

        }

        while(ch==1);

        getch();

}
```

Difference between containership and inheritance in C++

Containership: Containership is the phenomenon of using one or more classes within the definition of other class. When a class contains the definition of some other classes, it is referred to as composition, containment or aggregation. The data member of a new class is an object of some other class. Thus the other class is said to be composed of other classes and hence referred to as containership. Composition is often referred to as a "has-a" relationship because the objects of the composite class have objects of the composed class as members.

Inheritance: Inheritance is the phenomenon of deriving a new class from an old one. Inheritance supports code reusability. Additional features can be added to a class by deriving a class from it and then by adding new features to it. Class once written or tested need not be rewritten or redefined. Inheritance is also referred to as specialization or derivation, as one class is inherited or derived from the other. It is also termed as "is-a" relationship because every object of the class being defined is also an object of the inherited class.