C++ used to create general

* systems software,
* drivers for various computer devices,
* software for servers and software for specific applications
* creation of video games.
* applications that depend on direct hardware manipulation under real-time constraints.
* teach the basics of object-oriented features because it is simple and is also used in the fields of
* user interfaces and system files of Windows and Macintosh

C++ supports object-oriented programming (OOP), with four major principles of object-oriented development:

1. [Abstraction](https://www.w3schools.in/cplusplus-tutorial/data-abstraction/)
2. [Encapsulation](https://www.w3schools.in/cplusplus-tutorial/encapsulation/)
3. [Inheritance](https://www.w3schools.in/cplusplus-tutorial/inheritance/)
4. [Polymorphism](https://www.w3schools.in/cplusplus-tutorial/polymorphism/)

**Features of Object Oriented C++**

* The main focus remains on data rather than procedures.
* Object-oriented programs are segmented into parts called objects.
* Data structures are designed to categorize the objects.
* Data member and functions are tied together as a data structure.
* Data can be hidden and cannot be accessed by external functions using access specifier.
* Objects can communicate among themselves using functions.
* New data and functions can be easily added anywhere within a program whenever required.
* Since this is an object-oriented programming language, it follows a bottom up approach, i.e. the execution of codes starts from the main which resides at the lower section and then based on the member function call the working is done from the classes.

Object-oriented programming is a technique that provides a way of modularizing programs by creating memory area as a partition for both data and functions that can further be used as a template to create copies of modules on demand.

**STANDARD LIBERATRY IN C++**

C++ standard library was created after many years and it has three important parts:

1. C++ core language provides all the building blocks including [data types](https://www.w3schools.in/cplusplus-tutorial/data-types/), [variables](https://www.w3schools.in/cplusplus-tutorial/variables/), and literals etc.
2. The C++ Standard Library has a rich set of methods for manipulating files and strings.
3. The STL(Standard Template Library) provides a rich set of template classes for manipulating data structures.

**ANSI STANDARD FOR C++**

ANSI stands for American National Standard Institute & the ANSI standard began an attempt to ensure that C++ codes become portable - that code written for Microsoft's compiler will compile without having any errors can run on compilers of MAC or Linux or any other compiler. So, all major C++ compilers support the ANSI Standard.

The original C++ compiler was called **Cfront**. C++ is fast. Its speed can be attributed to its high-level features in conjunction with its low-level components.

**FEATURES OF I\O IN C++**

* C++ IO is type safe.
* C++ IO operations are based on streams of bytes and are device independent.

**I\O STREAMS IN C++**

C++ IO is based on streams, which are a sequence of bytes flowing in and out of the programs (just like water and oil flowing through a pipe). I/O systems in C++ are designed to work with a wide variety of devices including terminals, disks and tape drives. The IO system supplies an interface to the programmer that is independent of the actual device being accessed. This interface is known as a stream. A stream is a sequence of bytes which acts either as a source from which input data can be obtained or as a destination to which output data can be sent. The source stream which provides data to the program is called the input stream and the destination stream which receives output from the program is called the output stream.

To perform input and output, a C++ program must follow the steps mentioned below:

* Construct a stream object.
* Connect (Associate) the stream object to an actual IO device
* Perform input/output operations on the stream, via the functions defined in the stream's public interface in a device-independent manner.
* Disconnect (Dissociate) the stream to the actual IO device (e.g., close the file).
* Free the stream object.

**UNFORMATED I/O OPERATION**

cin and cout (pre-defined in iostream file) for input and output of data of various types. This has been made possible by overloading the operators << and >> to recognize all the basic C++ types.

* cin >>standard input stream
* cout <<standard output stream

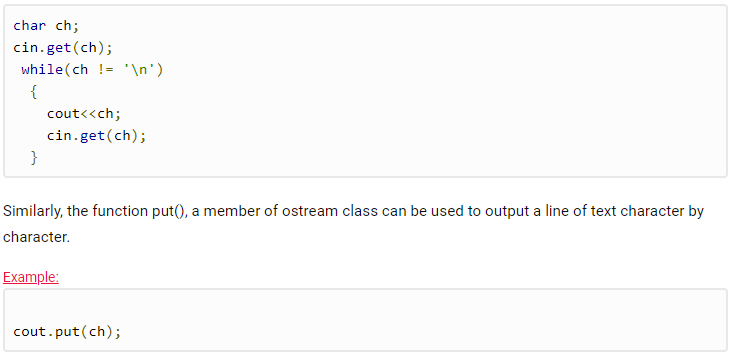
**put() and ge() function**

The classes istream and ostream defines two member functions get() and put() respectively to handle single character input/output operations.

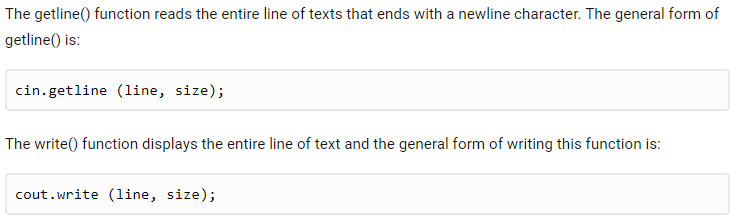
Get() function is of two types:

1. get(char \*)
2. get(void)

Both of them can be used to fetch a character including a blank space, tab or new-line character.



**Getline() & write()**



**MANIPULATORS**

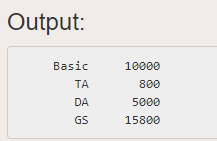
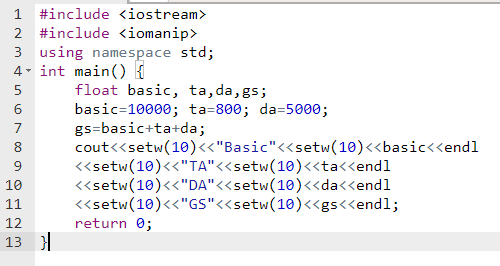
Manipulators are operators used in C++ for formatting output. The data is manipulated by the programmer's choice of display.

**Endl Manipulator**

Line feed operator. Point the cursor to the beginning of the next line. We can use \n (\n is an escape sequence) instead of **endl** for the same purpose.

**Setw Manipulator**

This manipulator sets the minimum field width on output, header file #include<iomanip>.



**Setfil Manipulator**

This is used by the setw manipulator. If a value does not entirely fill a field, then the character specified in the setfill argument of the manipulator is used for filling the fields.

#include <iostream>

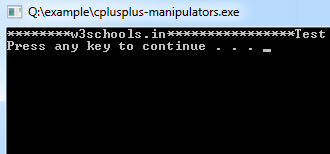
#include <iomanip>

using namespace std;

int main() {

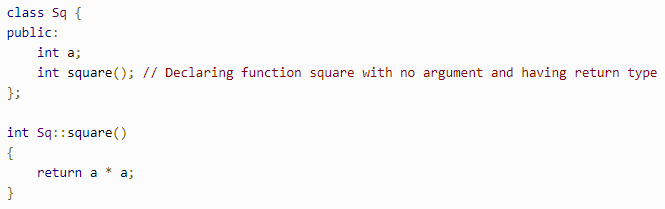
cout << setw(20) << setfill('\*') << "w3schools.in" << setw(20) << setfill('\*')<<"Test"<< endl;

}

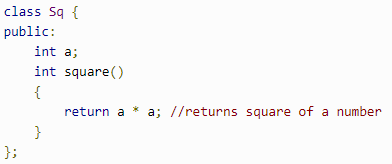


**MEMBER FUNCTION IN C++**

Member functions are C++ functions that have their declarations inside the class definition and these member functions work on the data member of the class. Member function definition can be written inside or outside the definition of the class. If the definition of the member function is inside the class definition, then it can define directly, but if it is defined outside the class then a special operator name scope resolution operator (::) is used along with the name of the class and the function name.



**or**

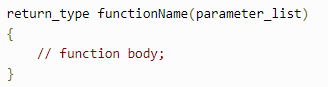


**TYPES OF MEMBER FUNCTIONS**

* Simple member function
* Static Member function
* Const function
* Inline function
* Friend function

**SIMPLE MEMBER FUNCTION**

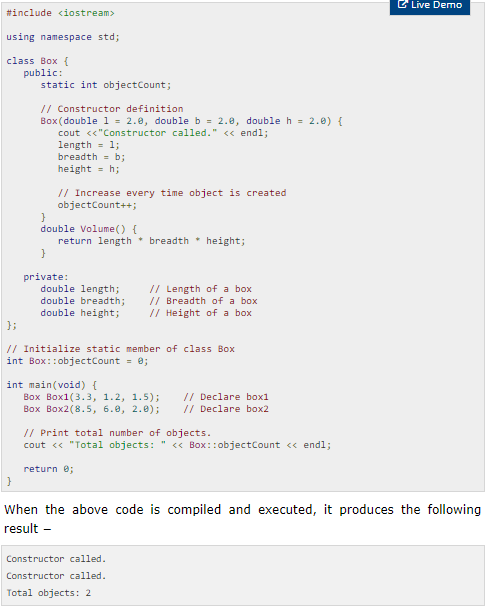
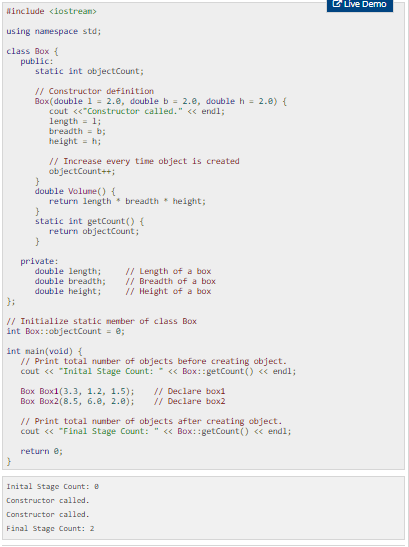
simple functions of C++ with or without return type and with or without parameters. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object. Member functions can be defined within the class definition or separately using **scope resolution operator**



**STATIC MEMBER FUNCTION**

The keyword 'static' is use. Static is mainly used to hold its positions. These functions work for the whole class rather than for a particular object of the class. The static member functions cannot access ordinary data members and member functions, but can only access the static data members and static member functions of a class. By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.

static member functions have a class scope and they do not have access to the **this** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.



**Constant member function**

Const keyword makes variable constant, which means once defined, their value cannot be changed. The basic syntax of const member function is:

void fun() const{}

**inline function**

When a function is declared as inline, the compiler places a copy of the code of that specific function at each point where the function is called at compile time.

**Friend Function**

Functions are declared as a friend using the keyword 'friend' to give private access to non-class functions. Programmers can declare a global function as a friend, or a member function of other class as the friend.

**CLASSES AND OBJECTS**

In object-oriented programming languages like C++, the data and functions (procedures to manipulate the data) are bundled together as a self-contained unit called an object. A class is an extended concept similar to that of structure in C programming language; this class describes the data properties alone. In C++ programming language, a class describes both the properties (data) and behaviors (functions) of objects. Classes are not objects, but they are used to instantiate objects.

**CLASS**

* A class is an abstract data type similar to 'C structure'.
* The Class representation of objects and the sets of operations that can be applied to such objects.
* The class consists of Data members and methods.

The primary purpose of a class is to hold data/information. This is achieved with attributes which are also known as data members.

The member functions determine the behavior of the class i.e. provide a definition for supporting various operations on data held in form of an object.

**CLASS DEFINATION**

Class class\_name

{

Data Members;

Methods;

}

EXAMPLE:

class A

{

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

}

* Private, Protected, Public is called visibility labels.
* The members that are declared private can be accessed only from within the class.
* Public members can be accessed from outside the class also.
* In C++, data can be hidden by making it private.

**Class members**

* **Data** and **functions** are members.
* Data Members and methods must be declared within the class definition.
* A member cannot be redeclared within a class.
* No member can be added elsewhere other than in the class definition.

Class A

{

int i;

int j;

void f (int, int); //**f** and **g** are a member function of **class A**. They determine the

int g(); //behavior of the objects of **class A**.

}

**Constructor and distructor**

C++ provides a special member function called the constructor which enables an object to initialize itself at the time of its creation. (automatic initialization of objects.). C++ also provides another member function called destructor which is used to destroy the objects when they are no longer required.

**CONSTRUCTOR**

Each time an instance of a class is created the constructor method is called. Constructors is a special member function of class and it is used to initialize the objects of its class. It is treated as a special member function because its name is the same as the class name. These constructors get invoked whenever an object of its associated class is created. It is named as "constructor" because it constructs the value of data member of a class. Initial values can be passed as arguments to the constructor function when the object is declared.

This can be done in two ways:

* By calling constructor explicitly
* By calling constructor implicitly

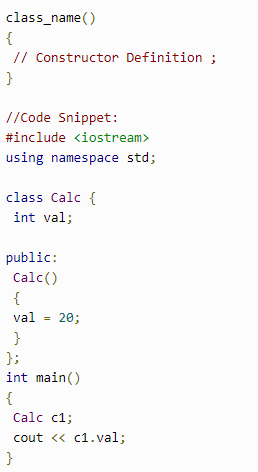
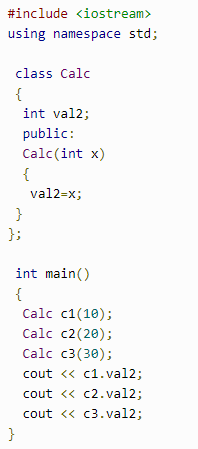
Special characteristics of Constructors:

* They should be declared in the public section
* They do not have any return type, not even void
* They get automatically invoked when the objects are created
* They cannot be inherited though derived class can call the base class constructor
* Like other functions, they can have default arguments
* You cannot refer to their address
* Constructors cannot be virtual

**TYPES OF CONSTRUCTOR**

1. Do nothing constructor
2. Default constructor
3. Parameterized constructor
4. Copy constructor

**Do nothing constructor:** Do nothing constructors are that type of constructor which does not contain any statements. Do nothing constructor is the one which has no argument in it and no return type.

**Default constructor:** the default constructor is the constructor which doesn't take any argument. It has no parameter but a programmer can write some initialization statement there A default constructor is very important for initializing object members, that even if we do not define a constructor explicitly, the compiler automatically provides a default constructor implicitly.

**Parameterized constructor:** A default constructor does not have any parameter, but programmers can add and use parameters within a constructor if required. This helps programmers to assign initial values to an object at the time of creation.

**Copy constructor:** constructor which takes an object as an argument and is used to copy values of data members of one object into another object. In this case, copy constructors are used to declaring and initializing an object from another object.

**DESTRUCTOR:**  destructors are used to destroy the objects that have been created by the constructor within the C++ program. Destructor names are same as the class name but they are preceded by a tilde (~). It is a good practice to declare the destructor after the end of using constructor

TASK TO BE DONE BEFORE TERMINATION OF OBJECT ARE WRITEN HERE.

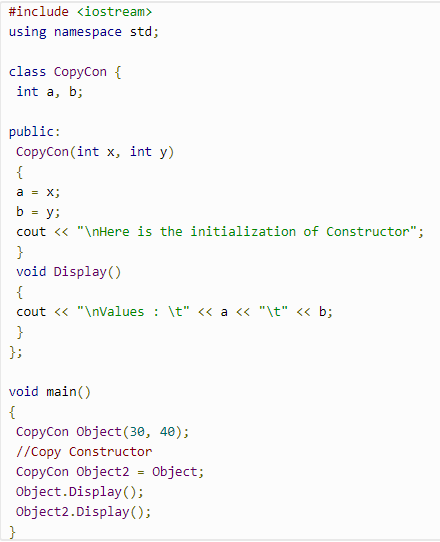
. Here's the basic declaration procedure of a destructor:

~Cube()

{

}

The destructor neither takes an argument nor returns any value and the compiler implicitly invokes upon the exit from the program for cleaning up storage that is no longer accessible.

**OVERLOADING:** more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively. 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**.

**Fundamentals of operator overloading:**

* C++ enables the programmer to overload most operators to be sensitive to the context in which they are used. The compiler generates an appropriate function or method call based on the operator's use.
* To overload an operator, write a function definition; the function name must be the keyword operator followed by the symbol for the operator being overloaded.
* To use an operator on class objects, that operator *must* be overloaded - with two exceptions. The assignment operator (=) may be used with two objects of the same class to perform a default memberwise assignment without overloading. The address operator (&) also can be used with objects of any class without overloading; it returns the address of the object in memory.
* The point of operator overloading is to provide the same concise expressive power for user-defined data types that C++ provides with its rich collection of operators that work on built-in types.
* Operator overloading is not automatic - the programmer must write operator overloading functions to perform the desired operations. Sometimes these functions are best made methods; sometimes they are best as friend functions.

resolve datatype of parameters in function overloading(3 methods compiler does it)

1)exact match

2)data type promotion

char,unsgined char, short =>int

float to double

3)standard conversion any =>any

**Function overloading:** 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.

**Operator overloading:** You can redefine or overload most of the built-in operators available. 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. Example:

complex operator+(const complex&);

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 –

complex operator+(const complex&, const complex&);

**Restriction on operator overloading**

* Following is the list of operators, which can not be overloaded –

|  |  |  |  |
| --- | --- | --- | --- |
| :: | .\* | . | ?: |

* The precedence and associativity of an operator (i.e., whether the operator is applied right-to-left or left-to-right) cannot be changed by overloading. Parentheses can be used to force the order of evaluation of overloaded operators in an expression.
* It is not possible to change the number of operands an operator takes: Overloaded unary operators remain unary operators, overloaded binary operators remain binary operators. C++'s only ternary operator, ?:, cannot be overloaded.
* It is not possible to create symbols for new operators; only existing operators can be overloaded.
* The meaning of how an operator works on built-in data types cannot be changed by overloading. The programmer cannot, for example, change the meaning of how + adds two integers. Operator overloading works only with objects of user-defined types or with a mixture of an object of a user-defined type and and a built-in type.
* There is no implicit overloading. Overloading an operator such as + does not automatically overload +=.

**ENCAPSULATION:** encapsulation can be defined as the process of hiding all of the details of an object that do not throw in or dealt with its essential characteristics. Encapsulation can also be defined as preventing access to non-essential details of classes or its objects. Encapsulation assists abstraction by providing a means of suppressing the non-essential details.

**Access modifiers :** Access specifiers are used to determining whether any other class or function can access member variables and functions of a particular class or within a program.

* Public: Here, the class members declared under public has to be obtainable and accessible to everyone. All those data members and member functions acknowledged as the public can be accessed by other classes also.
* Private: Private is used where no one can obtain or access the class members declared as private outside that class. When anyone tries to access or avail the private member of that class a compile-time error gets generated. By default, all members of a class remain in 'private' access mode.
* Protected: It is a special purpose access specifier which is similar to private but it creates class member inaccessible outside that class. But it can get accessed by any subclass of that class.

#include <iostream>

using namespace std;

class rectangle

{

   private:

   int l,b;

   public:

   rectangle(int x=2,int y=4)

   {

      l=x;

      b=y;

      cout<<"i am parametrized";

   }

   /\* rectangle()

   {

      cout<<"i am default";

   }\*/   void area()

   {

      cout<<"\narea is = "<<l\*b;

   }

};

int main()

{

   rectangle r;

   r.area();

   rectangle r1(3,6);

   r1.area();

   rectangle r2(10);

   r2.area();

}

**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

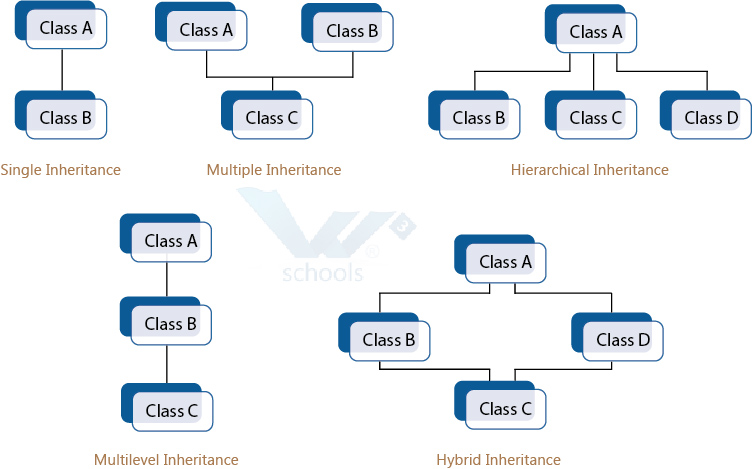
class derived-class: access-specifier base-class

 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.

|  |  |  |  |
| --- | --- | --- | --- |
| **Access** | **public** | **protected** | **private** |
| Same class | yes | yes | yes |
| Derived classes | yes | yes | no |
| Outside classes | yes | no | no |

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.

**Types of inheritance:**

* Single Inheritance
* Multiple Inheritance
* Hierarchical Inheritance
* Multilevel Inheritance
* Hybrid Inheritance (also known as Virtual Inheritance)

**Single inheritance:** In single inheritance, there is only one base class and one derived class. The Derived class gets inherited from its base class. This is the simplest form of inheritance.

**Multiple inheritance:** In this type of inheritance, a single derived class may inherit from two or more base classes

#include <iostream>

using namespace std;

class stud {

protected:

int roll, m1, m2;

public:

void get(){

cout << "Enter the Roll No.: "; cin >> roll;

cout << "Enter the two highest marks: "; cin >> m1 >> m2;

}

};

class extracurriculam {

protected:

int xm;

public:

void getsm(){

cout << "\nEnter the mark for Extra Curriculam Activities: "; cin >> xm;

}

};

class output : public stud, public extracurriculam {

int tot, avg;

public:

void display(){

tot = (m1 + m2 + xm);

avg = tot / 3;

cout << "\n\n\tRoll No : " << roll << "\n\tTotal : " << tot;

cout << "\n\tAverage : " << avg;

}

};

int main()

{

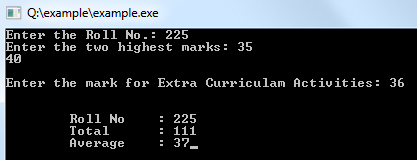
output O;

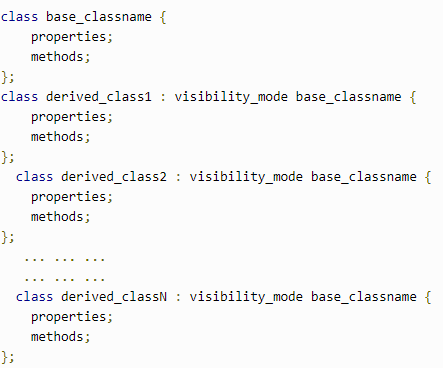
O.get();

O.getsm();

O.display();

}



**Hierarchical inheritance:** In this type of inheritance, multiple derived classes get inherited from a single base class.

#include <iostream>

#include <string.h>

using namespace std;

class member {

char gender[10];

int age;

public:

void get(){

cout << "Age: "; cin >> age;

cout << "Gender: "; cin >> gender;

}

void disp(){

cout << "Age: " << age << endl;

cout << "Gender: " << gender << endl;

}

};

class stud : public member {

char level[20];

public:

void getdata(){

member::get();

cout << "Class: "; cin >> level;

}

void disp2(){

member::disp();

cout << "Level: " << level << endl;

}

};

class staff : public member {

float salary;

public:

void getdata(){

member::get();

cout << "Salary: Rs."; cin >> salary;

}

void disp3(){

member::disp();

cout << "Salary: Rs." << salary << endl;

}

};

int main(){

member M;

staff S;

stud s;

cout << "Student" << endl;<< "Enter data" << endl;

s.getdata();

cout <<endl<< "Displaying data" << endl;

s.disp();

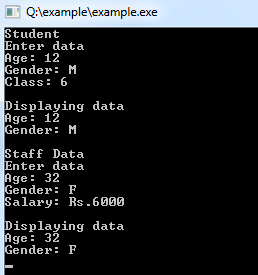
cout << endl<< "Staff Data" << endl;

cout << "Enter data" << endl;

S.getdata();

cout << endl<< "Displaying data" << endl;

}



**Const object:** when you want to create an object that you want no to be modified by any one. Some objects are regular object and other you want constant

When you have a constant object you can never call regular function , only constant function can be called

#include <iostream>

using namespace std;

class myConstfunc{

public:

void printfuc(){

cout<<"i am regular function "<<endl;

}

void printfunc() **const**{

cout<<"i am constant function "<<endl;

}

};

int main(){

cout << "Hello world!" << endl;

myConstfunc myfunc;

myfunc.printfuc();

**const** myConstfunc constobj;

constobj.printfunc();

return 0;

}

**This pointer:** Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

#include <iostream>

using namespace std;

class Box {

public:

// Constructor definition

Box(double l = 2.0, double b = 2.0, double h = 2.0) {

cout <<"Constructor called." << endl;

length = l;

breadth = b;

height = h;

}

double Volume() {

return length \* breadth \* height;

}

int compare(Box box) {

return this->Volume() > box.Volume();

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

int main(void) {

Box Box1(3.3, 1.2, 1.5); // Declare box1

Box Box2(8.5, 6.0, 2.0); // Declare box2

if(Box1.compare(Box2)) {

cout << "Box2 is smaller than Box1" <<endl;

} else {

cout << "Box2 is equal to or larger than Box1" <<endl;

}

return 0;

}

**o\p:**

Constructor called.

Constructor called.

Box2 is equal to or larger than Box1

**Preprocessor directives:** The preprocessors are the directives, which give instructions to the compiler to preprocess the information before actual compilation starts.

All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not C++ statements, so they do not end in a semicolon (;).

**There are 4 main types of preprocessor directives:**

1. Macros
2. File Inclusion
3. Conditional Compilation
4. Other directives

**MACRO**: The #define preprocessor directive creates symbolic constants. The symbolic constant is called a **macro** and the general form of the directive is −

#define macro-name replacement-text

#include <iostream>

using namespace std;

**#define PI 3.14159**

int main () {

cout << "Value of PI :" << PI << endl;

return 0;

}

**Function-like macro:** You can use #define to define a macro

#include <iostream>

using namespace std;

**#define MIN(a,b) (((a)<(b)) ? a : b) //use of ternary operator**

int main () {

int i, j;

i = 100;

j = 30;

cout <<"The minimum is " << MIN(i, j) << endl;

return 0;

}

**Op:** The minimum is 30

**CONDITIONAL COMPILATION:** construct is like the ‘if’ selection structure.

#ifndef NULL

#define NULL 0

#endif

 a program for debugging purpose. turn on or off the debugging using a single macro as−

#ifdef DEBUG

**cerr** <<"Variable x = " << x << endl;

#endif

This causes the **cerr** statement to be compiled in the program if the symbolic constant DEBUG has been defined before directive #ifdef DEBUG. You can use #if 0 statment to comment out a portion of the program as follows –

#if 0

code prevented from compiling

#endif

**## operator:** When CONCAT appears in the program, its arguments are concatenated and used to replace the macro.

#include <iostream>

using namespace std;

#define concat(a, b) a ## b

int main() {

int xy = 100;

cout << concat(x, y);

return 0;

}

**o/p:** 100

**predefined macros:**

**\_\_LINE\_\_** :This contains the current line number of the program when it is being compiled.

**\_\_FILE\_\_** :This contains the current file name of the program when it is being compiled.

**\_\_DATE\_\_ :**This contains a string of the form month/day/year that is the date of the translation of the source file into object code. **\_\_TIME\_\_ :**This contains a string of the form hour:minute:second that is the time at which the program was compiled.

#include <iostream>

using namespace std;

int main () {

cout << "Value of \_\_LINE\_\_ : " << \_\_LINE\_\_ << endl;

cout << "Value of \_\_FILE\_\_ : " << \_\_FILE\_\_ << endl;

cout << "Value of \_\_DATE\_\_ : " << \_\_DATE\_\_ << endl;

cout << "Value of \_\_TIME\_\_ : " << \_\_TIME\_\_ << endl;

return 0;

}

**o/p:**

Value of \_\_LINE\_\_ : 6

Value of \_\_FILE\_\_ : test.cpp

Value of \_\_DATE\_\_ : Feb 28 2011

Value of \_\_TIME\_\_ : 18:52:48

**FILE INCLUSION:** Both user and system header files are included using the preprocessing directive ‘#include’. It has two variants:

**#include <file>** This variant is used for system header files. It searches for a file named file in a standard list of system directories.

**#include "file"** This variant is used for header files of your own program. It searches for a file named file first in the directory containing the current file, then in the quote directories and then the same directories used for <file>.

**#undef Directive**: The #undef directive is used to undefine an existing macro. This directive works as:

#undef LIMIT

Using this statement will undefine the existing macro LIMIT. After this statement every “#ifdef LIMIT” statement will evaluate to false.

**SMA vs DMA**

SMA: static memory allocation (no relation with static key word)

DMA: dynamic memory allocation

SMA: declaration statement are for compiler to decide memory required to run this program during runtime execution. During execution of the program you cannot increase/decrease this memory .memory is fixed so called static memory allocation or compile time memory allocation. SMA members have a scope limit to its block and you can not change life of SMA variables. Everything is fixed; example life, memory, etc.

Good when we know how much memory we require , example program for adding 3 number so only 3 variable are needed. If we do not know how much number to add we use DMA.

DMA: variables are not made via declaration statements. We use **new** key word

Variables made with new keyword are dma variables they do not have name but only have address, new returns the address which we store in dma object/variable.

int \*p = new int;

float \*q = new float;

Complex \*ptr = new complex;

Float \*q = new float[5]

Int x ;

Cin>>x;

Int \*p = new int[x];

The **malloc()** function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

**Delete** keyword :to release memory that is assigned due to new keyword.

If dma variable created with new key word is not deleted, scope will be throughout the program unlike sma variable

Delete p;

Delete ptr;

**Virtual function:**base class pointer can point to the object of any of its descendent class but reverse is not true.

#include <iostream>

using namespace std;

class A{

public:

void f1(){cout<<"class a"<<endl;}

};

class B: public A{

public:

void f1(){ cout<<"class b"<<endl;} //function overriding

void f2(){}

};

int main () {

A \*p,o1;

B o2;

p = &o2;

o2.f1(); //op=> class b case 1

p->f1(); //op=> class a but it should be class b case 2

}

When a same object is use to call its functions it acts normally like case 1 but when parent class pointer points to its child class object and parent class pointer is used to call functions in child class things go abnormal like case 2

Compiler identities correct version of the function and binds it to its call, this process is called early binding or compile time binding. To do this compiler checks the datatype of the object which has called the function

O2 type =>child class so o2.f1 = class b

P pointer => parent class so p->f1() = class a due to early binding

Incase 2 due to early binding compiler do not know the p pointer points which address so compiler doesn’t know p has address of o1 or o2 as memory location is not yet assigned at compile time. So the early binding is done the bases of type of pointer which was decided at the time of declaration. This is problem in function overriding. And the solution is not early bind rather late bind i.e. binding is done on runtime or dynamic binding. So virtual keyword is used so at function call rather refereeing type of pointer, we will refer content of pointer

#include <iostream>

using namespace std;

class A{

public:

**virtual** void f1(){cout<<"class a"<<endl;}

};

class B: public A{

public:

void f1(){ cout<<"class b"<<endl;} //function overriding

void f2(){}

};

int main () {

A \*p,o1;

B o2;

p = &o2;

o2.f1(); //op=> class b case 1

p->f1(); //op=> class a but it should be class b case 2

}

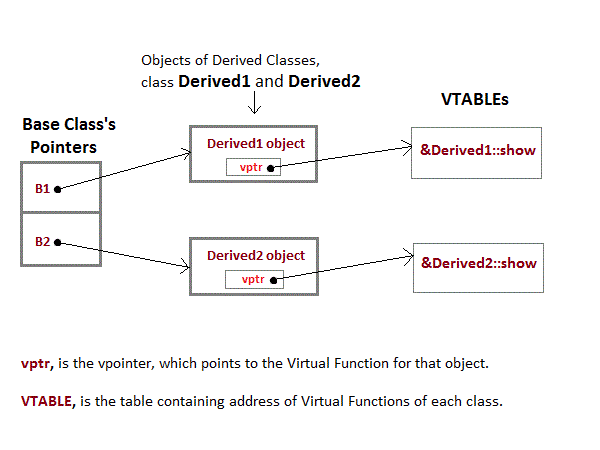
**Op:** class b

class b

just mention virtual key word in parent class with overriding do not mention virtual key word again

#### **Mechanism of Late Binding**

To accomplich late binding, Compiler creates **VTABLEs**, for each class with virtual function. The address of virtual functions is inserted into these tables. Whenever an object of such class is created the compiler secretly inserts a pointer called **vpointer**, pointing to VTABLE for that object. Hence when function is called, compiler is able to resovle the call by binding the correct function using the vpointer.



#### Important Points to Remember

1. Only the Base class Method's declaration needs the **Virtual** Keyword, not the definition.
2. If a function is declared as **virtual** in the base class, it will be virtual in all its derived classes.
3. The address of the virtual Function is placed in the **VTABLE** and the copiler uses **VPTR**(vpointer) to point to the Virtual Function.

**Pure virtual function:** a do nothing function is a pure virtual function i.e. virtual function with no definition. we cannot create object of base class. The class inheriting this class must override pure virtual function. A base class function with no definition must be a virtual if not, it can be accessed via pointer ,parent class pointer pointing child class and the function (with no definition) is called creates due to early binding an error is there.

## class Base{ //Abstract base class

## public:

## **virtual void show() = 0;** //Pure Virtual Function

## };

## class Derived:public Base{

## public:

## void **show**() //overriding

## { cout << "Implementation of Virtual Function in Derived class"; }

## };

## int main(){

## Base obj; //Compile Time Error

## Base \*b;

## Derived d;

## b = &d;

## b->show();

## } **op: Implementation of Virtual Function in Derived class**

**Virtual destructor:** Destructors in the Base class can be Virtual. Whenever Upcasting is done, Destructors of the Base class must be made virtual for proper destrucstion of the object when the program exits.

**NOTE :** Constructors are never Virtual, only Destructors can be Virtual.

Lets first see what happens when we do not have a virtual Base class destructor.

class Base

{

public:

~Base() {cout << "Base Destructor\t"; }

};

class Derived:public Base

{

public:

~Derived() { cout<< "Derived Destructor"; }

};

int main()

{

Base\* b = new Derived; //Upcasting

delete b;

}

**Output :** Base Destructor

In the above example, **delete b** will only call the Base class destructor, which is undesirable because, then the object of Derived class remains undestructed, because its destructor is never called. Which results in memory leak.

#### **Upcasting with Virtual Destructor**

Now lets see. what happens when we have Virtual destructor in the base class.

class Base

{

public:

**virtual** ~Base() {cout << "Base Destructor\t"; }

};

class Derived:public Base

{

public:

~Derived() { cout<< "Derived Destructor"; }

};

int main()

{

Base\* b = new Derived; //Upcasting

delete b;

}

**Output :** Derived Destructor

Base Destructor

When we have Virtual destructor inside the base class, then first Derived class's destructor is called and then Base class's destructor is called, which is the desired behaviour.

#### Pure Virtual Destructors

* Pure Virtual Destructors are legal in C++. Also, pure virtual Destructors must be defined, which is against the pure virtual behaviour.
* The only difference between Virtual and Pure Virtual Destructor is, that pure virtual destructor will make its Base class Abstract, hence you cannot create object of that class.
* There is no requirement of implementing pure virtual destructors in the derived classes.

class Base

{

public:

**virtual** ~Base() = 0; //Pure Virtual Destructor

};

**Base::~Base()** { cout << "Base Destructor"; } //Definition of Pure Virtual Destructor

class Derived:public Base

{

public:

~Derived() { cout<< "Derived Destructor"; }

};

**Abstract class:** Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

#### Characteristics of Abstract Class

1. Abstract class cannot be instantiated, but pointers and refrences of Abstract class type can be created.
2. Abstract class can have normal functions and variables along with a pure virtual function.
3. Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

#### Why can't we create Object of Abstract Class ?

When we create a pure virtual function in Abstract class, we reserve a slot for a function in the VTABLE(studied in last topic), but doesn't put any address in that slot. Hence the VTABLE will be incomplete.

As the VTABLE for Abstract class is incomplete, hence the compiler will not let the creation of object for such class and will display an errror message whenever you try to do so.

**If u do not want to define pure virtual function then in derived class you can again make that function as pure virtual function , then class inheriting that derived class will have to define that class and object of the 1st derived cannot be created. In 2nd derived class you can again make that function a pure virtual function or override it**

**To generalize is concept in object oriented principles this is the reason to make abstract class.**

**Template :**template keyword to define function template and template classes. It is the way to make your function or class generalize as far as data type is concen.

|  |  |
| --- | --- |
| **Function template:**  #include <iostream>  using namespace std;  template <class X> X big(X a,X b){  if(a>b)  return a;  else  return b;  }  int main(){  cout<<big(4,5)<<endl;  cout<<big(3.3,4.6);  } | **Class template:**  #include<conio.h>  #include <iostream>  using namespace std;  **template<class X>** class ArrayList{  private:  struct ControlBlock{  int capacity;  X \*arr\_ptr;  };  ControlBlock \*s;  public:  ArrayList(int capacity){  s = new ControlBlock;  s->capacity = capacity;  s->arr\_ptr = new X[s->capacity];  }  void addElement(int index, X data){  if (index>=0&&index <s->capacity)  s->arr\_ptr[index] = data;  else  cout<<"invalid index"<<endl;  }  void viewList(){  for(int i = 0; i<s->capacity;i++)  cout<<s->arr\_ptr[i]<<" ";  }  };  int main(){  ArrayList <float>l(4);  l.addElement(0,3.2);  l.addElement(2,4.1);  l.addElement(3,3.2);  l.addElement(1,2.01);  l.viewList();  } |

You may overload a function template either by a non-template function or by another function template.

#include <iostream>

using namespace std;

template<class T> void f(T x, T y) { cout << "Template" << endl; }

void f(int w, int z) { cout << "Non-template" << endl; }

int main() {

f( 1 , 2 ); f('a', 'b'); f( 1 , 'b');

}

**Op:**Non-template Template Non-template

**Template non-type arguments :** non-type template argument provided within a template argument list is an expression whose value can be determined at compile time. Such arguments must be constant expressions, addresses of functions or objects with external linkage, or addresses of static class members. Non-type template arguments are normally used to initialize a class or to specify the sizes of class members.

For non-type integral arguments, the instance argument matches the corresponding template parameter as long as the instance argument has a value and sign appropriate to the parameter type.

For non-type address arguments, the type of the instance argument must be of the form identifier or &identifier, and the type of the instance argument must match the template parameter exactly, except that a function name is changed to a pointer to function type before matching.

The resulting values of non-type template arguments within a template argument list form part of the template class type. If two template class names have the same template name and if their arguments have identical values, they are the same class

template<class T, int size> class Myfilebuf

{

T\* filepos;

static int array[size];

public:

Myfilebuf() { /\* ... \*/ }

~Myfilebuf();

advance(); // function defined elsewhere in program

};

Myfilebuf <double,200>x;

|  |
| --- |
| **Using Inheritance Between Templates** |
| template <class QueueItem> class Queue  {  private:  QueueItem buffer[100];  int head, tail, count;  public:  Queue();  void Insert(QueueItem item);  QueueItem Remove();  ~Queue();  };  template <class QueueItem> class InspectableQueue : public Queue<QueueItem>  {  public:  InspectableQueue();  QueueItem Inspect(); // return without removing the first element  ~InspectableQueue();  }; |

**Exception handling:** An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,**and **throw**.

* **throw** − A program throws an exception when a problem shows up. This is done using a **throw** keyword.
* **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
* **try** − A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks

 A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code

|  |  |
| --- | --- |
| try {  // protected code  } catch( ExceptionName e1 ) {  // catch block  } catch( ExceptionName e2 ) {  // catch block  } catch( ExceptionName eN ) {  // catch block  } | You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations. |

Errors can be broadly categorized into two types. We will discuss them one by one.

1. Compile Time Errors
2. Run Time Errors

**Compile Time Errors** – Errors caught during compiled time is called Compile time errors. Compile time errors include library reference, syntax error or incorrect class import.

**Run Time Errors** - They are also known as exceptions. An exception caught during run time creates serious issues.

Standard Exceptions in C++

* **std::exception** - Parent class of all the standard C++ exceptions.
* **logic\_error** - Exception happens in the internal logical of a program.
  + **domain\_error** - Exception due to use of invalid domain.
  + **invalid argument** - Exception due to invalid argument.
  + **out\_of\_range** - Exception due to out of range i.e. size requirement exceeds allocation.
  + **length\_error** - Exception due to length error.
* **runtime\_error** - Exception happens during runtime.
  + **range\_error** - Exception due to range errors in internal computations.
  + **overflow\_error** - Exception due to arithmetic overflow errors.
  + **underflow\_error** - Exception due to arithmetic underflow errors
* **bad\_alloc** - Exception happens when memory allocation with new() fails.
* **bad\_cast** - Exception happens when dynamic cast fails.
* **bad\_exception** - Exception is specially designed to be listed in the dynamic-exception-specifier.
* **bad\_typeid** - Exception thrown by typeid.

#include <iostream>

#include<conio.h>

using namespace std;

int main() {

int x[3]={-1,2,};

for(int i=0;i<2;i++) {

int ex=x[i];

try {

if (ex > 0)

throw ex;

else

throw 'ex';

}

catch (int ex) {

cout << " Integer exception" ;

}

catch (const char\* ex) {

cout << " Character exception" ;

}

catch (...) {

cout << "Special exception";

}

}

getch();

return 0;

}

**Op:** Integer exception Integer exception

**Namespaces:** Consider a situation, when we have two persons with the same name, Zara, in the same class. Whenever we need to differentiate them definitely we would have to use some additional information along with their name, like either the area, if they live in different area or their mother’s or father’s name, etc.

Same situation can arise in your C++ applications. For example, you might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.

A **namespace** is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope.

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows −

namespace namespace\_name {

// code declarations

}

To call the namespace-enabled version of either function or variable, prepend (::) the namespace name as follows −

name::code; // code could be variable or function.

#include <iostream>

using namespace std;

// first name space

namespace first\_space {

void func() {

cout << "Inside first\_space" << endl;

}

}

// second name space

namespace second\_space {

void func() {

cout << "Inside second\_space" << endl;

}

}

int main () {

// Calls function from first name space.

first\_space::func();

// Calls function from second name space.

second\_space::func();

return 0;

}

**Op:** Inside first\_space

Inside second\_space

## **The using directive**

You can also avoid prepending of namespaces with the **using namespace** directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace.

#include <iostream>

using namespace std;

// first name space

namespace first\_space {

void func() {

cout << "Inside first\_space" << endl;

}

}

// second name space

namespace second\_space {

void func() {

cout << "Inside second\_space" << endl;

}

}

using namespace first\_space;

int main () {

// This calls function from first name space.

func();

return 0;

}**op:** Inside first\_space

The ‘using’ directive can also be used to refer to a particular item within a namespace. For example, if the only part of the std namespace that you intend to use is cout, you can refer to it as follows −

using std::cout;

Subsequent code can refer to cout without prepending the namespace, but other items in the **std**namespace will still need to be explicit as follows −

#include <iostream>

using std::cout;

int main () {

cout << "std::endl is used with std!" << std::endl;

return 0;

}

If we compile and run above code, this would produce the following result −

std::endl is used with std!

Names introduced in a **using** directive obey normal scope rules. The name is visible from the point of the **using** directive to the end of the scope in which the directive is found. Entities with the same name defined in an outer scope are hidden.

## **Discontiguous Namespaces**

A namespace can be defined in several parts and so a namespace is made up of the sum of its separately defined parts. The separate parts of a namespace can be spread over multiple files.

So, if one part of the namespace requires a name defined in another file, that name must still be declared. Writing a following namespace definition either defines a new namespace or adds new elements to an existing one −

namespace namespace\_name {

// code declarations

}

## **Nested Namespaces**

Namespaces can be nested where you can define one namespace inside another name space as follows −

namespace namespace\_name1 {

// code declarations

namespace namespace\_name2 {

// code declarations

}

}

You can access members of nested namespace by using resolution operators as follows −

// to access members of namespace\_name2

using namespace namespace\_name1::namespace\_name2;

// to access members of namespace:name1

using namespace namespace\_name1;

In the above statements if you are using namespace\_name1, then it will make elements of namespace\_name2 available in the scope as follows −

#include <iostream>

using namespace std;

// first name space

namespace first\_space {

void func() {

cout << "Inside first\_space" << endl;

}

// second name space

namespace second\_space {

void func() {

cout << "Inside second\_space" << endl;

}

}

}

using namespace first\_space::second\_space;

int main () {

// This calls function from second name space.

func();

return 0;

}

**Op:**Inside second\_space

# Why can't we overload the assignment operator using the friend function?

You need to understand the way assignment operator works, the right hand operand is source and lefthand operand is the target of assignment, the left hand operand is typically called the LValue and right hand operand is called RValue. Now, LValue must be a variable and cannot be constant, RValue can be a variable as well as a constant.

When you overload assignment operator as member function it is the LValue object for which that assignment operator is called and the RValue object is passed as parameter and if the RValue is a constant and you have provided a constructor to convert that constant to object then using that constructor that particular constant (which was a RValue) would be converted to object and will passed as parameter to assignment operator.

Now if you provide a constant as LValue and say you are allowed to overload assignment operator as friend function then both the operands will be passed to the friend function as parameter, now if there is a constructor that receives a constant and creates object, then compiler will use that constructor and convert that LValue (which is a constant for the given scenario) to object and will make a call to the assignment operator and you will be able to modify a constant in that way, which will be completely illogical and illegal.

Hence, to avoid modifying constant as LValue, assignment operator was restricted to be overloaded as friend function.

## **What is a virtual base class?**

- An ambiguity can arise when several paths exist to a class from the same base class. This means that a child class could have duplicate sets of members inherited from a single base class.  
- C++ solves this issue by introducing a virtual base class. When a class is made virtual, necessary care is taken so that the duplication is avoided regardless of the number of paths that exist to the child class.

## **What is Virtual base class? Explain its uses.**

- When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class. This can be achieved by preceding the base class’ name with the word virtual.  
- Consider the following example :

class A   
{   
   public:   
       int i;   
};  
  
class B : virtual public A   
{   
   public:   
       int j;   
};  
  
class C: virtual public A   
{   
   public:   
       int k;   
};  
  
class D: public B, public C   
{   
   public:   
       int sum;   
};  
  
int main()   
{   
   D ob;   
   ob.i = 10; //unambiguous since only one copy of i is inherited.   
   ob.j = 20;   
   ob.k = 30;   
   ob.sum = ob.i + ob.j + ob.k;   
   cout << “Value of i is : ”<< ob.i<<”\n”;   
   cout << “Value of j is : ”<< ob.j<<”\n”; cout << “Value of k is :”<< ob.k<<”\n”;   
   cout << “Sum is : ”<< ob.sum <<”\n”;   
  
   return 0;   
}

# **Rethrowing an exception (C++ only)**

If a catch block cannot handle the particular exception it has caught, you can rethrow the exception. The rethrow expression (throwwithout *assignment\_expression*) causes the originally thrown object to be rethrown.

Because the exception has already been caught at the scope in which the rethrow expression occurs, it is rethrown out to the next dynamically enclosing try block. Therefore, it cannot be handled by catch blocks at the scope in which the rethrow expression occurred. Any catch blocks for the dynamically enclosing try block have an opportunity to catch the exception.

The following example demonstrates rethrowing an exception:

#include <iostream>

using namespace std;

struct E {

const char\* message;

E() : message("Class E") { }

};

struct E1 : E {

const char\* message;

E1() : message("Class E1") { }

};

struct E2 : E {

const char\* message;

E2() : message("Class E2") { }

};

void f() {

try {

cout << "In try block of f()" << endl;

cout << "Throwing exception of type E1" << endl;

E1 myException;

throw myException;

}

catch (E2& e) {

cout << "In handler of f(), catch (E2& e)" << endl;

cout << "Exception: " << e.message << endl;

throw;

}

catch (E1& e) {

cout << "In handler of f(), catch (E1& e)" << endl;

cout << "Exception: " << e.message << endl;

throw;

}

catch (E& e) {

cout << "In handler of f(), catch (E& e)" << endl;

cout << "Exception: " << e.message << endl;

throw;

}

}

int main() {

try {

cout << "In try block of main()" << endl;

f();

}

catch (E2& e) {

cout << "In handler of main(), catch (E2& e)" << endl;

cout << "Exception: " << e.message << endl;

}

catch (...) {

cout << "In handler of main(), catch (...)" << endl;

}

}[copy to clipboard](javascript:void(0);)

The following is the output of the above example:

In try block of main()

In try block of f()

Throwing exception of type E1

In handler of f(), catch (E1& e)

Exception: Class E1

In handler of main(), catch (...)[copy to clipboard](javascript:void(0);)

The try block in the main() function calls function f(). The try block in function f() throws an object of type E1 named myException. The handler catch (E1 &e) catches myException. The handler then rethrows myException with the statement throw to the next dynamically enclosing try block: the try block in the main() function. The handler catch(...) catches myException.

**overload << && >>**

#include <iostream>

#include<conio.h>

using namespace std;

class complex{

private:

int a,b;

public:

void setData(int x,int y){a=x;b=y;}

friend ostream& operator<<(ostream&,complex);

friend istream& operator>>(istream&,complex&);

};

ostream& operator<<(ostream &dout,complex c){

cout<<"a "<<c.a<<" "<<"b "<<c.b<<endl;

return dout;

}

istream& operator>>(istream &din,complex &c){

cin>>c.a>>c.b;

return din;

}

int main()

{

complex c1;

cin>>c1;

cout<<"you entered ::"<<endl;

cout<<c1;

return 0;

}

**copy constructor:**

compex(complex &c){

a=c.a;b=c.b;

}

void main(){

complex c4(c1);

}

**friend function**

#include<iostream>

#include<conio.h>

using namespace std;

class B;

class A{

private:

int a;

public:

friend void fun(A,B);

void setData(int y){a=y;}

};

class B{

private:

int b;

public:

void setData(int y){b=y;}

friend void fun(A,B);

};

void fun(A obj1,B obj2){

cout<<"sum is "<<obj1.a+obj2.b;

}

int main()

{

A obj1;

obj1.setData(5);

B obj2;

obj2.setData(6);

fun(obj1,obj2);

return 0;

}

**overload + operator with friend function**

#include <iostream>

#include<conio.h>

using namespace std;

class complex{

private:

int a,b;

public:

void setData(int x,int y){a=x;b=y;}

void showData(){cout<<"a: "<<a<<endl<<"b: "<<b<<endl;}

friend complex operator + (complex,complex);

};

complex operator+(complex X,complex Y){

complex temp;

temp.a = X.a+Y.a;

temp.b = X.b+ Y.b;

return temp;

}

int main()

{

complex c1,c2,c3;

c1.setData(3,4);

c2.setData(5,6);

c3 = c1+c2;

c3.showData();

return 0;

}

**overload unary - with freind**

#include <iostream>

#include<conio.h>

using namespace std;

class complex{

private:

int a,b;

public:

void setData(int x,int y){a=x;b=y;}

void showData(){cout<<"a: "<<a<<endl<<"b: "<<b<<endl;}

friend complex operator-(complex);

};

complex operator-(complex X){

complex temp;

temp.a = -X.a;

temp.b = -X.b;

return temp;

}

int main()

{

complex c1,c2,c3;

c1.setData(3,4);

c2.setData(5,6);

// c3 = c1.operator-();

c3 = -c1;// caller object is c1;way to implement polymorphism

c3.showData();

return 0;

}

**operator overloading**

#include <iostream>

#include<conio.h>

using namespace std;

class complex{

private:

int a,b;

public:

void setData(int x,int y){a=x;b=y;}

void showData(){cout<<"a: "<<a<<endl<<"b: "<<b<<endl;}

complex operator+(complex c){

complex temp;

temp.a = a+c.a;

temp.b = b+ c.b;

return temp;

}

};

int main()

{

complex c1,c2,c3;

c1.setData(3,4);

c2.setData(5,6);

// c3 = c1.operator+(c2);

c3 = c1+c2 // caller object is c1;way to implement polymorphism

c3.showData();

return 0;

}

**overloading unary operator**

#include <iostream>

#include<conio.h>

using namespace std;

class complex{

private:

int a,b;

public:

void setData(int x,int y){a=x;b=y;}

void showData(){cout<<"a: "<<a<<endl<<"b: "<<endl;}

complex operator-(){

complex temp;

temp.a = -a;

temp.b = -b;

return temp;

}

};

int main()

{

complex c1,c2,c3;

c1.setData(3,4);

c2.setData(5,6);

// c3 = c1.operator-();

c3 = -c1;// caller object is c1;way to implement polymorphism

c3.showData();

return 0;

}

**perincriment operator overloading**

#include <iostream>

#include<conio.h>

using namespace std;

class complex{

private:

int a,b;

public:

void setData(int x,int y){a=x;b=y;}

void showData(){cout<<"a: "<<a<<endl<<"b: "<<b<<endl;}

complex operator++(){

complex temp;

temp.a = ++a;

temp.b = ++b;

return temp;

}

};

int main()

{

complex c1,c2;

c1.setData(3,4);

c2 = ++c1;

c2.showData();

return 0;

}

**post incriment operator overloading**

#include <iostream>

#include<conio.h>

using namespace std;

class complex{

private:

int a,b;

public:

void setData(int x,int y){a=x;b=y;}

void showData(){cout<<"a: "<<a<<endl<<"b: "<<b<<endl;}

complex operator++(int){

complex temp;

temp.a = a++;

temp.b = b++;

return temp;

}

};

int main()

{

complex c1,c2;

c1.setData(3,4);

c2 = c1++;

c2.showData();

cout<<"c1"<<endl;

c1.showData();

return 0;

}

polymorphysiom :

1)function overloading =>compile time polymorphisom

2)operator overloading =>compile time polymorphisom

3)virtual function =>runtime time polymorphisom

structure is collection of dissimilar elements

structure is a way to group variables

structure is used to create data type

in c++ structure can also have functions in it

in c++ access modifer is there in structure

struct book{

int book\_id;

char title[20];

float price;

};

in c++ using data type made with structure it is not mandatory to use keey word struct

struct book b1; or book b1 = {10,"c++",300.00};

b1.book\_id = 101;

strcpy(b2.tiitle,"manan ki c++");

b1=b2;

-------------------------------------------------------------------------------------------

in c++ structure all members are default public unlike class where members are default private (only difference in class and structure )

object state is collection of values of object member

static local variable

static member variable:

declared inside the class

qualified with static keyword

also known as class member variable/class variable

do not belong to object but to whole class

there will be onle one copy of static member variable for whole class

any object can use same copy of class variable

can also be use with class name

if object of class does not exist these class variable will exist

after class you need to define static member variable

class Account{

private:

int balance;

static float roi;

public:

void setBalance(int b){

balance = b;}

};

float Account::roi=3.5f;

void main(){

Account a1,a2;

}

static member variable:

declared inside the class4

qualified with static keyword

also known as class member function/class function

do not belong to object but to whole class

can also be use with class name

if object of class does not exist these class variable will exist

these function can only access static menber variable

class Account{

private:

int balance;

static float roi;

public:

void setBalance(int b){

balance = b;}

static void setRoi(float r)

{roi=r;}

};

float Account::roi=3.5f;

void main(){

Account a1,a2;

Account::setRoi(4.5f);

}