OBJECT ORIENTED PROGRAMMING (OOP) USING C++

A QUESTION BANK

Chinmay D.Bhamare

2014

Its my first Object Oriented Programming (OOP) language Question Bank. In this notes all types of question are available related to c++. Most of the questions are related to BCA syllabus.

So enjoy this notes and make your study easy

- chinmay D. Bhamare

(Smt.S.M.Agrrawal Inst.Of Mgt.,Chalisgaon)

Write Ans. Of Following Question .

What is class?

The class is one of the defining ideas of object-oriented programming. Among the important ideas about classes are:

- A class can have subclasses that can inherit all or some of the characteristics of the class. In relation to each subclass, the class becomes the superclass.
- Subclasses can also define their own methods and variables that are not part of their superclass.
- The structure of a class and its subclasses is called the class hierarchy.

Data Types ?

Boolean –bool
Character -char
Integer -int
Floating point- float
Double floating point- double
Valueless- void
Wide Character –wchar_t

What is Object ?

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

```
Box Box1;          // Declare Box1 of type Box
Box Box2;          // Declare Box2 of type Box
```

Use of Delete operator ?

Using the **delete** operator on an object deallocates its memory. A program that dereferences a pointer after the object is deleted can have unpredictable results or crash.

When **delete** is used to deallocate memory for a C++ class object, the object's destructor is called before the object's memory is deallocated (if the object has a destructor).

If the operand to the **delete** operator is a modifiable l-value, its value is undefined after the object is deleted.

```
int* set = new int[100];
//use set[]
delete [] set;
```

what is mean by function?

Functions allow to structure programs in segments of code to perform individual tasks.

In C++, a function is a group of statements that is given a name, and which can be called from some point of the program. The most common syntax to define a function is:

```
type name ( parameter1, parameter2, ...) { statements }
```

What is mean by inline function?

C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line. A function definition in a class definition is an inline function definition, even without the use of the **inline**specifier.

Types of Inheritance ?

Inheritance those are provided by C++ are as followings:

1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

Use of structure in C++ ?

As we have seen, a structure is a good way of storing related data together. It is also a good way of representing certain types of information. Complex numbers in mathematics inhabit a two dimensional plane (stretching in real and imaginary directions). These could easily be represented here by

```
typedef struct {
        double real;
        double imag;
} complex;
```

doubles have been used for each field because their range is greater than floats and because the majority of mathematical library functions deal with doubles by default.

In a similar way, structures could be used to hold the locations of points in multi-dimensional space. Mathematicians and engineers might see a storage efficient implementation for sparse arrays here.

Apart from holding data, structures can be used as members of other structures. Arrays of structures are possible, and are a good way of storing lists of data with regular fields, such as databases.

Another possibility is a structure whose fields include pointers to its own type. These can be used to build chains (programmers call these linked lists), trees or other connected structures. These are rather daunting to the new programmer, so we won't deal with them here.

Types of Operators in C++ ?

Arithmetic operators (+, -, *, /, %)
Assignment operator (=)
Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)
Increment and decrement (++, --)
Relational and comparison operators (==, !=, >, <, >=, <=)
Logical operators (!, &&, ||)
Conditional ternary operator (?)
Bitwise operators (&, |, ^, ~, <<, >>)

What do you mean by oops ?

Object-oriented programming (OOP) is a programming language model organized aroundobjects rather than "actions" and data rather than logic. Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data.The programming challenge was seen as how to write the logic, not how to define the data. Object-oriented programming takes the view that what we really care about are the objects we want

to manipulate rather than the logic required to manipulate them. One of the principal advantages of object-oriented programming techniques over procedural programming techniques is that they enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its featuresfrom existing objects. This makes object-oriented programs easier to modify.

What is Cunstructor ?

Constructor is specialized method that is contained in a class. Constructor will be triggered automatically when an object is created. Purpose of the Constructors is to initialize an object of a class.

Constructor's name is identical to the Class name. Constructors can take parameters but constructors have no return type. This is because; constructors return an object by itself.

```
class Example
{
     int x, y;
   public:
     Example();
     Example(int a, int b);
//parameterized constructor
};
Example :: Example()
{
}
Example :: Example(int a, int b)
{
     x = a;
     y = b;
}
```

What Is Polymorphism ?

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Input And Out Put Statement in C++?

when the computer gets the data from the keyboard , the user is said to be acting interactively.
Putting data into variables using **cin** and the operator **>>**.The **syntax** of cin together with >> is

cin>>variable;

if two variables then

cin>>variable1>>variable2;

This is called an input statement. In c++ , >> is called the stream extraction operator.

By this way for so on variables ...

suppose that statement.

int feet;
int inches;

then input is

cin>>feet>>inches;

Output statements

In c++ output on standard output device is use **cout** and the operator **<<** . The syntax for output statement is

cout<< expression;

This is called an output statement. In C++ , **<<** is called the **stream insertion** operator.

Syntax while and Do while Statement in c++ ?

```
While Loop:
while(condition)
{
   statement(s);
}
```

```
Do While :
do
```

```
{
   statement(s);
}while( condition );
```

If else Statement Define Syntax?

```
if(boolean_expression)
{
   // statement(s) will execute if the boolean expression is true
}
else
{
  // statement(s) will execute if the boolean expression is false
}
```

What is encapsulation ?

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**

Data encapsulation is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

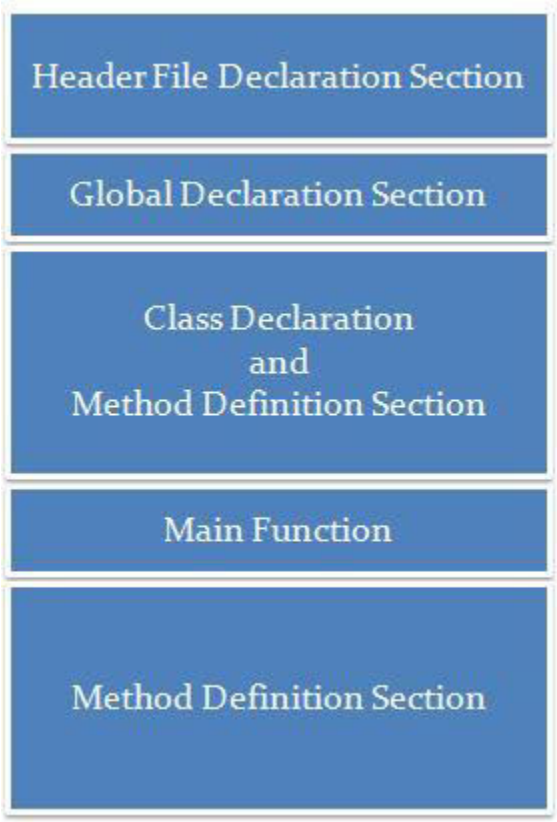
C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected** and **public**members. By default, all items defined in a class are private.

Explain Structure of C++ Program ?

C++ Programming language is most popular language after C Programming language.

C++ is first Object oriented programming language.We have summarize structure of

C++ Program in the following Picture -



Structure of C++ Program

Section 1 : Header File Declaration Section

1. Header files used in the program are listed here.
2. Header File provides **Prototype declaration** for different library functions.
3. We can also include **user define header file**.
4. Basically all preprocessor directives are written in this section.

Section 2 : Global Declaration Section

1. Global Variables are declared here.
2. Global Declaration may include -
    - Declaring Structure
    - Declaring Class
    - Declaring Variable

Section 3 : Class Declaration Section

1. Actually this section can be considered as sub section for the global declaration section.
2. Class declaration and all methods of that class are defined here.

Section 4 : Main Function

1. Each and every C++ program always starts with main function.
2. This is entry point for all the function. Each and every method is called indirectly through main.
3. We can create class objects in the main.
4. Operating system call this function automatically.

Section 5 : Method Definition Section

1. This is optional section . Generally this method was used in C Programming

New Operator ?

If you prefix **new** with the scope resolution operator (`::`), the global **operator new()** is used. If you specify an *`argument_list`*, the overloaded **new**operator that corresponds to that *`argument_list`* is used. The *`type`* is an existing built-in or user-defined type. A *`new_type`* is a type that has not already been defined and can include type specifiers and declarators.

An allocation expression containing the **new** operator is used to find storage in free store for the object being created. The *new expression* returns a pointer to the object created and can be used to initialize the object. If the object is an array, a pointer to the initial element is returned.

You can use `set_new_handler()` only to specify what **new** does when it fails.

You cannot use the **new** operator to allocate function types, **void**, or incomplete class types because these are not object types. However, you can allocate pointers to functions with the **new** operator. You cannot create a reference with the **new** operator.

When the object being created is an array, only the first dimension can be a general expression. All subsequent dimensions must be constant integral expressions. The first dimension can be a general expression even when an existing *`type`* is used. You can create an array with zero bounds with the **new**operator. For example:

```
char * c = new char[0];
```

An object created with **`operator new()`** or **`operator new[]()`**

Manipulator In c++ ?

Manipulators are functions specifically designed to be used in conjunction with the insertion (<<) and extraction (>>) operators on stream objects, for example:

```
cout << boolalpha;
```

They are still regular functions and can also be called as any other function using a stream object as argument, for example:

```
boolalpha (cout);
```

Manipulators are used to change formatting parameters on streams and to insert or extract certain special characters.

Use Of Scope resolution Operator ?

The `::` (scope resolution) operator is used to qualify hidden names so that you can still use them. You can use the unary scope operator if a namespace scope or global scope name is hidden by an explicit declaration of the same name in a block or class. For example:

```
int count = 0;

int main(void) {
  int count = 0;
  ::count = 1;  // set global count to 1
  count = 2;    // set local count to 2
  return 0;
}
```

The declaration of `count` declared in the `main()` function hides the integer named `count` declared in global namespace scope. The statement `::count = 1`accesses the variable named `count` declared in global namespace scope.

You can also use the class scope operator to qualify class names or class member names. If a class member name is hidden, you can use it by qualifying it with its class name and the class scope operator.

Syntax Of Structure ?

struct structure _name

{

data_type member1;

data_type member2;

---------- ----------;

---------- ----------;

} ; //Semicolon is must & Before semicolon you can declare structure

What Is Syntax Of Inline Function ?

```
#include <iostream>

using namespace std;
```

```
inline int Max(int x, int y)
{
   return (x > y)? x : y;
}

// Main function for the program
int main( )
{

   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) << endl;
   return 0;
}
```

Class Syntax ?

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...

} object_names;
```

Questions In Briefly :

Define Polymorphism Static & Dynamic polymorphism?

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Static polymorphism refers to an entity existing in different physical forms simultaneously. Static polymorphism involves binding of functions based on the number, type, and sequence of arguments. The various types of parameters are specified in the function declaration, and therefore the function can be bound to calls at compile time. This form of association is called *early binding*. The term *early binding* stems from the fact that when the program is executed, the calls are already bound to the appropriate functions.

The resolution of a function call is based on number, type, and sequence of arguments declared for each form of the function. Consider the following function declaration:

```
void add(int , int);
void add(float, float);
```

When the *add()* function is invoked, the parameters passed to it will determine which version of the function will be executed. This resolution is done at compile time.

Dynamic Polymorphism

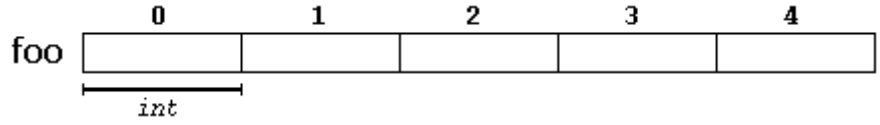
Dynamic polymorphism refers to an entity changing its form depending on the circumstances. A function is said to exhibit dynamic polymorphism when it exists in more than one form, and calls to its various forms are resolved dynamically when the program is executed. The term *late binding* refers to the resolution of the functions at run-time instead of compile time. This feature increases the flexibility of the program by allowing the appropriate method to be invoked, depending on the context.

Define Array? single And Multidimensional array ?

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

That means that, for example, five values of type `int` can be declared as an array without having to declare 5 different variables (each with its own identifier). Instead, using an array, the five `int` values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index.

For example, an array containing 5 integer values of type `int` called `foo` could be represented as:



where each blank panel represents an element of the array. In this case, these are values of type `int`. These elements are numbered from 0 to 4, being 0 the first and 4 the last; In C++, the first element in an array is always numbered with a zero (not a one), no matter its length.

Single Dimensional Array

By definition, an array is collection of data that is contiguous in memory. Like any variable, an array must be declared before it is used. There are also numerous types of arrays. The first type is the standard 1 dimensional array. A 1D array contains 1 long row of data.

type **name[capacity];**

Multi Dimensional Array

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y, you would write something as follows:

```
type arrayName [ x ][ y ];
```

Where **type** can be any valid C++ data type and **arrayName** will be a valid C++ identifier.
A two-dimensional array can be think as a table, which will have x number of rows and y number of columns. A 2-dimensional array **a**,

Multiple Inheritance with example ?

Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes.

The constructors of inherited classes are called in the same order in which they are inherited.

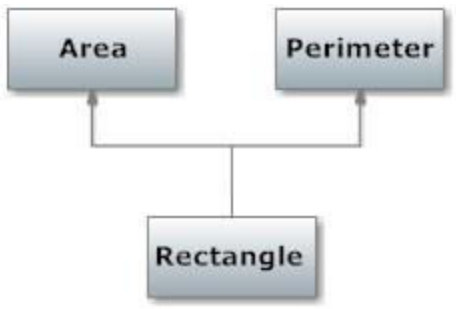


Figure: Multiple Inheritance Example

Friend Function In brief ?

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword **friend** as follows:

```
class Box
{
   double width;
public:
   double length;
   friend void printWidth( Box box );
   void setWidth( double wid );
};
```

To declare all member functions of class ClassTwo as friends of class ClassOne, place a following declaration in the definition of class ClassOne:

```
friend class ClassTwo;
```

Pure Virtual function in brief ?

A **pure virtual function** is a function that has ***the notation "= 0"*** in the declaration of that function. Why we would want a pure virtual function and what a pure virtual function looks like is explored in more detail below.

Here is a simple example of what a pure virtual function in C++ would look like:

Simple Example of a pure virtual function in C++

```
class SomeClass {
public:
```

```
    virtual void pure_virtual() = 0;  // a pure virtual function
    // note that there is no function body
};
```

The **"= 0"** portion of a pure virtual function is also known as the *pure specifier*, because it's what makes a pure virtual function "pure". Although the pure specifier appended to the end of the virtual function definition may look like the function is being assigned a value of 0, that is **not** true. The notation "= 0" is just there to indicate that the virtual function is a pure virtual function, and that the function has no body or definition. Also note that we named the function "pure_virtual" – that was just to make the example easier to understand, but it certainly does not mean that all pure virtual functions must have that name since they can have any name they want.

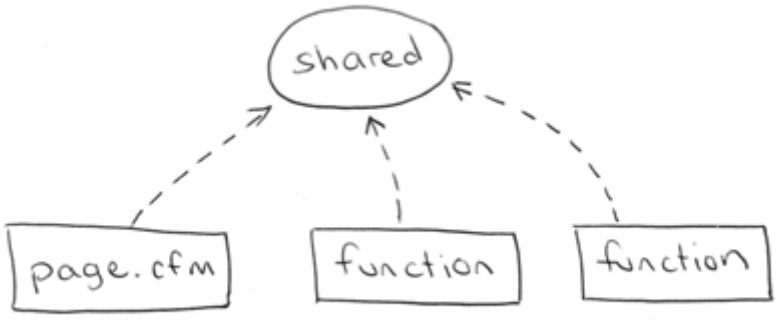
Distinction between object oriented programming and Procedure oriented programming.?

Procedural programming

In procedural programing our code is organised into small "procedures" that use and change our data. In ColdFusion, we write our procedures as either custom tags or functions. These functions typically take some input, do something, then produce some output. Ideally your functions would behave as "black boxes" where input data goes in and output data comes out.

The key idea here is that our functions have no intrinsic relationship with the data they operate on. As long as you provide the correct number and type of arguments, the function will do its work and faithfully return its output.

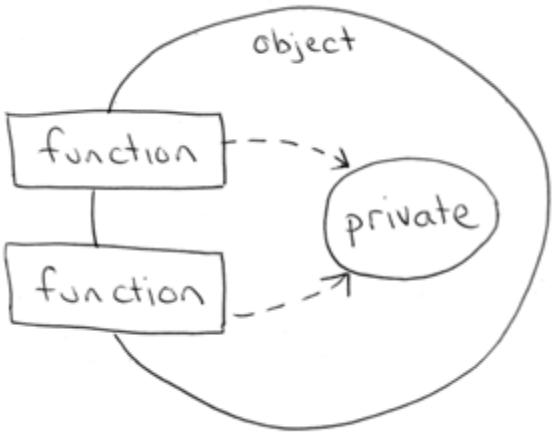
Sometimes our functions need to access data that is not provided as a parameter, i.e., we need access data that is outside the function. Data accessed in this way is considered "global" or "shared" data.



So in a procedural system our functions use data they are "given" (as parameters) but also directly access any shared data they need.

Object oriented programming

In object oriented programming, the data and related functions are bundled together into



onlу be manipulated by calling the
cked away inside your objects and
mething with that data. In a well

designed object oriented system objects never access shared or global data, they are only permitted to use the data they have, or data they are given.

Explain different file operations ?

Data File Handling In C++

File. The information / data stored under a specific name on a storage device, is called a file.
Stream. It refers to a sequence of bytes.

Text file. It is a file that stores information in ASCII characters. In text files, each line of text is terminated with a special character known as EOL (End of Line) character or delimiter character. When this EOL character is read or written, certain internal translations take place.
Binary file. It is a file that contains information in the same format as it is held in memory. In binary files, no delimiters are used for a line and no translations occur here.

Classes for file stream operation

ofstream: Stream class to write on files
ifstream: Stream class to read from files
fstream: Stream class to both read and write from/to files.

Opening a file

OPENING FILE USING CONSTRUCTOR

ofstream *fout*(*"results"*); //output only
ifstream *fin*(*"data"*); //input only

OPENING FILE USING open()

Stream-object.open(*"filename"*, mode)

```
ofstream ofile;
ofile.open("data1");

ifstream ifile;
ifile.open("data2");
```

Closing File

```
fout.close();
fin.close();
```

INPUT AND OUTPUT OPERATION

put() and get() function

the function put() writes a single character to the associated stream. Similarly, the function get() reads a single character form the associated stream.

example :

file.get(ch);
file.put(ch);

write() and read() function

write() and read() functions write and read blocks of binary data.

example:

file.read((char *)&*obj*, sizeof(*obj*));
file.write((char *)&*obj*, sizeof(*obj*));

ERROR HANDLING FUNCTION

FUNCTION	RETURN VALUE AND MEANING
eof()	returns true (non zero) if end of file is encountered while reading; otherwise return false(zero)
fail()	return true when an input or output operation has failed
bad()	returns true if an invalid operation is attempted or any unrecoverable error has occurred.
good()	returns true if no error has occurred.

File Pointers And Their Manipulation

All i/o streams objects have, at least, one internal stream pointer:
ifstream, like istream, has a pointer known as the get pointer that points to the element to be read in the next input operation.
ofstream, like ostream, has a pointer known as the put pointer that points to the location where the next element has to be written.

Finally, fstream, inherits both, the get and the put pointers, from iostream (which is itself derived from both istream and ostream).

These internal stream pointers that point to the reading or writing locations within a stream can be manipulated using the following member functions:

seekg()	moves get pointer(input) to a specified location
seekp()	moves put pointer (output) to a specified location
tellg()	gives the current position of the get pointer

tellp()	gives the current position of the put pointer

The other prototype for these functions is:
seekg(offset, refposition);
seekp(offset, refposition);
The parameter offset represents the number of bytes the file pointer is to be moved from the location specified by the parameter refposition. The refposition takes one of the following three constants defined in the ios class.

ios::beg	start of the file
ios::cur	current position of the pointer
ios::end	end of the file

example:

file.seekg(-10, ios::cur);

Note on Visibility mode ?

Access Modifiers :- These are also Called as Access Visibility Controls means they defined where a method and Data Member of class will be used either inside a class ,outside a class ,in inherited class or in main Method They Tells us the Scope of Methods where they would be used Various types of Access Modifiers are as follows:-

Public Access: - Public Access modifiers Specifies that data Members and Member Functions those are declared as public will be visible in entire class in which they are defined. Public Modifier is used when we wants to use the method any where either in the class or from outside the class. The Variables or methods those are declared as public are accessed in any where , Means in any Class which is outside from our main program or in the inherited class or in the class that is outside from our own class where the method or variables are declared.

2) **Protected Access**:- The Methods those are declared as Protected Access modifiers are Accessible to Only in the Sub Classes but not in the Main Program , This is the Most important Access Modifiers which is used for Making a Data or Member Function as he

may only be Accessible to a Class which derives it but it doesn't allows a user to Access the data which is declared as outside from Program Means Methods those are Declared as Protected are Never be Accessible to Another Class The Protected will be Accessible to Only Sub Class and but not in your Main Program.

3) **Private Access**:- The Methods or variables those are declared as private Access modifiers are not would be not Accessed outside from the class or in inherited Class or the Subclass will not be able to use the Methods those are declared as Private they are Visible only in same class where they are declared. By default all the Data Members and Member Functions is Private, if we never specifies any Access Modifier in front of the Member and Data Members Functions.

Unary Operator in brief ?

A unary operator, in C++, is an operator that takes a single operand in an expression or a statement. The unary operators in c++ are +, -,!, ~, ++, -- and the cast operator.

The signature of the declaration of a unary operator includes the operator token and the type of parameter; it does not require the return type and the name of the parameter.

All the C# unary operators have predefined implementation that will be used by default in an expression. These unary operators can be overloaded in user-defined types with custom implementation by defining static member functions using the "operator" keyword.

- Unary Plus Operator (+): The result of an operation on a numeric type is the value of the operand itself. This operator has been predefined for all numeric types.
- Unary Minus Operator (-): This operator can be used to negate numbers of the integer, floating-point and decimal type.
- Logical Complement (negation) Operator (!): This operator can be used only with operands of Boole type.

- Bitwise Complement (negation) Operator (~): This operator can be used with integer, unit, long and ulong operand types. The result of the operation is a bitwise complement (inverse of the binary representation) of the operand.
- Prefix Increment (++) and Decrement (--) Operator: The operand can be a variable, property access, or an indexer access. With an increment operator, the result of the operation for operands of integer type would be the value incremented by 1. With a decrement operator, the result would be the value decremented by 1 from the operand. The increment/decrement operator can also be used with postfix notation
- Cast Operator: Used to build cast expressions for conversion to a given type. This operator is represented by the symbol, "T," where T is the type to which the operand or the result of the expression must be converted

Explain nesting of classes ?

A *nested class* is declared within the scope of another class. The name of a nested class is local to its enclosing class. Unless you use explicit pointers, references, or object names, declarations in a nested class can only use visible constructs, including type names, static members, and enumerators from the enclosing class and global variables.

Member functions of a nested class follow regular access rules and have no special access privileges to members of their enclosing classes. Member functions of the enclosing class have no special access to members of a nested class.

```
class A {
  int x;

  class B { };

  class C {

    // The compiler cannot allow the following
    // declaration because A::B is private:
    //   B b;

    int y;
    void f(A* p, int i) {

    // The compiler cannot allow the following
    // statement because A::x is private:
    //   p->x = i;
```

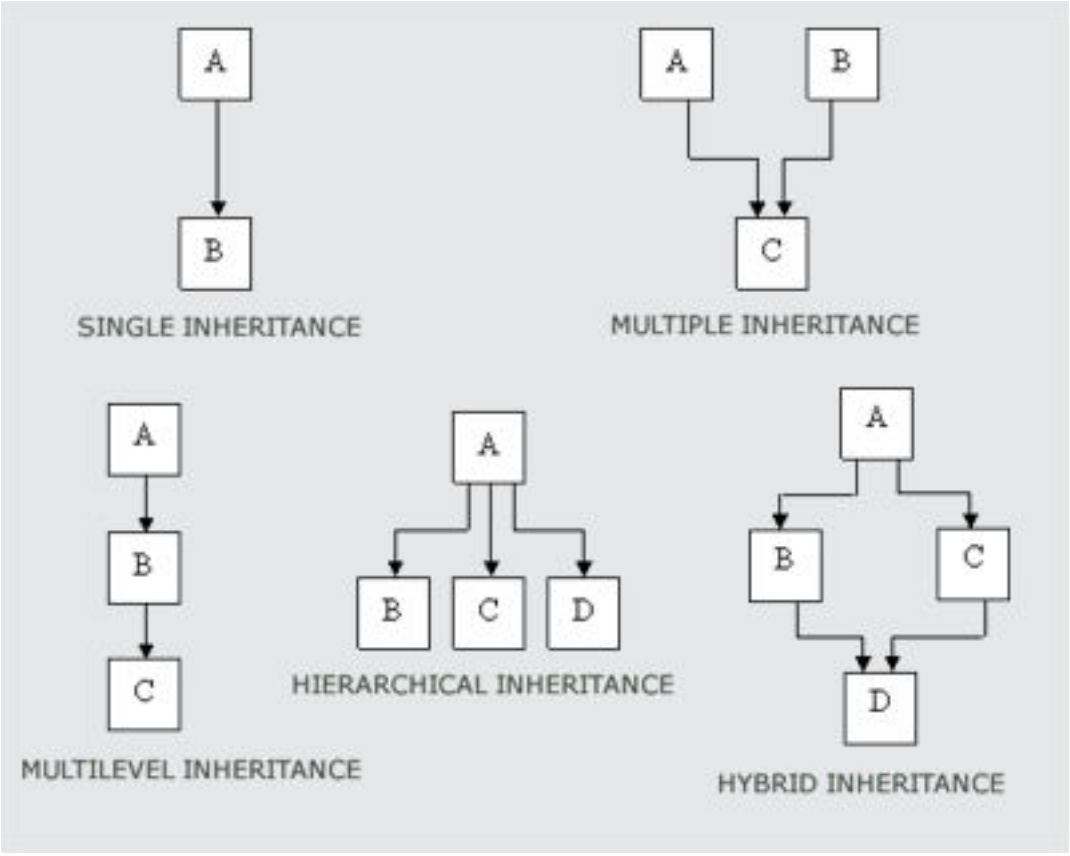
```
    }
  };

  void g(C* p) {

    // The compiler cannot allow the following
    // statement because C::y is private:
    //   int z = p->y;
  }
};

int main() { }
```

The compiler would not allow the declaration of object `b` because class `A::B` is private. The compiler would not allow the statement `p->x = i` because `A::x`is private. The compiler would not allow the statement `int z = p->y` because `C::y` is private.

You can define member functions and static data members of a nested class in namespace scope. For example, in the following code fragment, you can access the static members `x` and `y` and member functions `f()` and `g()` of the nested class `nested` by using a qualified type name. Qualified type names allow you to define a `typedef` to represent a qualified class name. You can then use the `typedef` with the `::` (scope resolution) operator to refer to a nested class or class member,

Types Of inheritance with example ?

Single Inheritance: It is the inheritance hierarchy wherein one derived class inherits from one base class.
Multiple Inheritance: It is the inheritance hierarchy wherein one derived class inherits from multiple base class(es)
Hierarchical Inheritance: It is the inheritance hierarchy wherein multiple subclasses inherit from one base class.
Multilevel Inheritance: It is the inheritance hierarchy wherein subclass acts as a base class for other classes.
Hybrid Inheritance: The inheritance hierarchy that reflects any legal combination of other four types of inheritance.



Difference between structure and Union ?

Union:

If we are having the less memory to use in our program, for example 64K, we can use a single memory location for more than one variable this is called union.

You can use the unios in the followig locations.

You can share a single memory location for a variable myVar1 and use the same location for myVar2 of different data type when myVar1 is not required any more.

You can use it if you want to user, for example, a long variable as two short type variables.

When you dont know what type of data is to be passed to a function, and you pass union which contains all the possible data types

```
1. union myUnion{
2.   int var1;
3.   long var2;
4. };
```

Structure:

A structure is a convenient tool for handling a group of logically related data items. Structure help to organize complex data is a more meaningful way. It is powerful concept that we may after need to use in our program Design.

A structure is combination of different data types. Lets take the exmple of a book, if we cant to declare a book we will be thinking about the name, title, authors and publisher of the book and publishing year. So to declare a book we need to have some complex data type which can deal with more than one data types. Lets declare a Book.

We can declare a structure to accommodate the book.

```
5. struct Book
6. {
7.   char Name[100];
8.   char Author[100];
9.   char Publisher[80];
10.         int Year;
11. };
```

Explain destructor with example ?

A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Following example explains the concept of destructor:

```
#include <iostream>

using namespace std;
```

```
class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line();
      ~Line();

   private:
      double length;
};

 Line::Line(void)
{
    cout << "Object is being created" << endl;
}
Line::~Line(void)
{
    cout << "Object is being deleted" << endl;
}

void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}

int main( )
{
   Line line;

   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```


What you mean by Operator over loading ?

In C++ the overloading principle applies not only to functions, but to operators too. That is, of operators can be extended to work not just with built-in types but also classes. A programmer can provide his or her own operator to a class by overloading the built-in operator to perform some

specific computation when the operator is used on objects of that class. Is operator overloading really useful in real world implementations? It certainlly can be, making it very easy to write code that feels natural (we'll see some examples soon). On the other hand, operator overloading, like any advanced C++ feature, makes the language more complicated. In addition, operators tend to have very specific meaning, and most programmers don't expect operators to do a lot of work, so overloading operators can be abused to make code unreadable. But we won't do that.

```
Complex a(1.2,1.3);      //this class is used to represent
complex numbers
Complex b(2.1,3);        //notice the construction taking 2
parameters for the real and imaginary part
```

What is Static member ?with example?

We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.
A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator **::** to identify which class it belongs to.

```
#include <iostream>

using namespace std;

class Box
{
   public:
      static int objectCount;
      // 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;
         // Increase every time object is created
         objectCount++;
      }
      double Volume()
      {
```

```
         return length * breadth * height;
      }
      static int getCount()
      {
         return objectCount;
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main(void)
{

   // Print total number of objects before creating object.
   cout << "Inital Stage Count: " << Box::getCount() << endl;

   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   // Print total number of objects after creating object.
   cout << "Final Stage Count: " << Box::getCount() << endl;

   return 0;
}
```

Difference between constructor and destructor ?

Constructor:

- Constructor is Used to Initialize the Object.
- Constructor can takes arguments.
- Constructor overloading can be possible means more than one constructors can be defined insame class.
- Constructor has same name as class name.
- Syntax of constructor:

```
class class_name
 {
```

```
    clas_sname(){}
    class_name(argulist){}
} ;
```

- Constructor are of following:
  1)Default Constructor.
  2)Parameterized Constructor.
  3)Copy Constructor.

- Constructors can be used to dynamically initialize the memory.
- Constructor indirectly use the New operator to initialize the object.

Destructor:

- Destructor is used to destroy the object that are created in memory previously.
- Destructor can not take any arguments.
- Destructor overloading can not be possible.
- Destructor has same name as class name with tiled operator.
- Syntax of Destructor:

```
class class_name
 {

    ~class-name(void){}
 };
```

- Destructor has no any types.
- Destructor can be used to deallocate the memory.
- Destructor indirectly use the Delete operator to destroy the object initialize by constructor.

Note on pointer ?

Pointers are variables that contain memory addresses (see Addresses, Pointers, and References). They are an essential data type in C and C++, and are used for the following:

- Array variables are pointers and pointers can be used as alternative to subscripts.
- Dynamic memory allocation/deallocation returns a pointer to the allocated memory.
- For efficiency by passing the address of an object instead of copying the object to a function. Reference parameters typically used.
- For function parameters that will be changed by the function (*out* or *inout* parameters). Reference parameters typically used.

```
int* ip;  // declares ip to be a pointer to an int.
```

- You will also see variations in the spacing such as

```
int *ip;
int * ip;
```

Note on String ?

The **string** class is part of the C++ standard library. A string represents a sequence of characters.

To use the string class, #include the header file:

```
#include <string>
```

The standard `string` class provides support for such objects with an interface similar to that of standard containers, but adding features specifically designed to operate with strings of characters.

The `string` class is an instantiation of the basic_string class template that uses `char` as the character type, with its default char_traits and allocator types (see basic_string for more info on thetemplate).

Types of operator overloading ?

There are two types of operator overloading:
1. Unary operator overloading
2. Binary operator overloading

1. Unary operator overloading
Unary operators are the ones that operate on one operand, one such operator is the unary minus (-) operator which is used to change the sign of the operand it acts upon.

This operator works well with basic data types such as int, float, etc.. In this section we will see how to overload this operator so that it can work the same way for user defined data types as it does for basic data types, i.e. change the sign of the operand.
The unary minus operator function does not take any arguments but it changes the sign of the data members of the object that calls this function. Since this function is a member function of the same class, it can directly access the members of the object that calls it.

```
friend void operator -(space &s); //declaration
void operator -(space &s) //definition
{
s.x = -s.x;
s.y = -s.y;
s.z = -s.z;
}
```

2. Binary operator overloading
Binary operators can be overloaded in a similar manner as unary operators.

We should note the following features of an operator function for a binary operator:
It receives only one class type argument explicitly, in case of a member function. For a friend function, two class types are received as arguments.
It returns a class type.

```
user_defined operator+(user_defined rhs)
{
user_defined temp;
temp.x = x + rhs.x;
temp.y = y + rhs.y;
return temp;
}
```

Application of C++?

It is a versatile language for handling very large programs
It is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real life application systems
It allows us to create hierarchy-related objects, so we can build special object-oriented libraries which can be used later by many programmers
While C++ is able to map the real-world problem properly, the C part of C++ gives the language the ability to get close to the machine – level details
C++ programs are easily maintainable and expandable

Features of OOPs ?

1. Programs are divided into objects
2. Data structures designed such that they characterize the objects.

3. Functions that operate on the data of an object are tied together in the data structure
4. Data is hidden and cannot be accessed by external functions
5. Objects may communicate with each other through functions
6. New data and functions can be easily added whenever necessary
7. IT follows bottom-up approach in program design
8. Concentration is on data rather than procedure

What are control flow statement ?

When a program is run, the CPU begins execution at the top of main(), executes some number of statements, and then terminates at the end of main(). The sequence of statements that the CPU executes is called the program's path. Straight-line programs have sequential flow — that is, they take the same path (execute the same statements) every time they are run (even if the user input changes).

However, often this is not what we desire. For example, if we ask the user to make a selection, and the user enters an invalid choice, ideally we'd like to ask the user to make another choice. This is not possible in a straight-line program.

Fortunately, C++ provides control flow statements (also called flow control statements), which allow the programmer to change the CPU's path through the program. There are quite a few different types of control flow statements.

Different C++ flow control Statements

- if statement
- else if construct
- switch statement
- break statement
- while loop
- do while loop
- for loop

Explain access control ?

Private

Private is the default access specifier for every declared data item in a class. Private data belongs to the same class in which it is created and can only be used by the other members of the same class.

Protected
When a data item is declared as protected it is only accessible by the derived class member.

Public
Public allows to use the declared data item used by anyone from anywhere in the program. Data items declared in public are open to all and can be accessed by anyone willing to use their values and functions they provide.

What is abstract class ?

In programming languages, an abstract class is a generic class (or type of object) used as a basis for creating specific objects that conform to its protocol, or the set of operations it supports. Abstract classes are not instantiated directly.

Abstract classes are useful when creating hierarchies of classes that model reality because they make it possible to specify an invariant level of functionality in some methods, but leave the implementation of other methods until a specific implementation of that class (a derived class) is needed.

```
Example :
class AB {
public:
  virtual void f() = 0;
};
```

An abstract class has at least one abstract method. An abstract method will not have any code in the base class; the code will be added in its derived classes. The abstract method in the derived class should be implemented with the same access modifier, number and type of argument, and with the same return type as that of the base class. Objects of abstract class type cannot be created, because the code to instantiate an object of the abstract class type will result in a compilation error.

Merits and Demerits of OOP ?

OOP stands for object oriented programming. It offers many benefits to both the developers and the users. Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater

programmer productivity, better quality of software and lesser maintenance cost. The primary advantages are:

Merits:

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple objects to coexist without any interference.
- It is possible to map objects in the problem domain to those objects in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in an implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects make the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.
- Polymorphism can be implemented i.e. behavior of functions or operators or objects can be changed depending upon the operations.

Demerits:

- It requires more data protection.
- Inadequate for concurrent problems
- Inability to work with existing systems.
- Compile time and run time overhead.
- Unfamiliraity causes training overheads.

What is function Prototype ?

A function prototype is a declaration in C and C++ of a function, its name, parameters and return type. Unlike a full definition, the prototype terminates in a semi-colon.

```
int getsum(float * value) ;
```

Prototypes are used in header files so that external functions in other files can be called and the compiler can check the parameters during compilation.

Explain pointer arithmathic with example ?

As you understood pointer is an address which is a numeric value; therefore, you can perform arithmetic operations on a pointer just as you can a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmatic operation on the pointer:

```
ptr++
```

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main ()
{
   int  var[MAX] = {10, 100, 200};
   int  *ptr;

   // let us have array address in pointer.
   ptr = var;
   for (int i = 0; i < MAX; i++)
   {
      cout << "Address of var[" << i << "] = ";
      cout << ptr << endl;

      cout << "Value of var[" << i << "] = ";
      cout << *ptr << endl;

      // point to the next location
      ptr++;
   }
   return 0;
}
```

What is recursive function ?

Technically, a recursive function is a function that makes a call to itself. To prevent infinite recursion, you need an if-else statement (of some sort) where one branch makes a recursive call, and the other branch does not. The branch without a recursive call is usually the base case (base cases do not make recursive calls to the function).

Functions can also be mutually recursive. For example, function **f**() can call function **g**() and function **g**() can call function **f**(). This is still considered recursion because a function can eventually call itself. In this case,**f**() indirectly calls itself.Functions can be tail-recursive. In a tail-recursive function, none of the recursive call do additional work after the recursive call is complete (additional work includes printing, etc), except to return the value of the recursive call.The following is typical of a tail-recursive function return.

```
   return rec_func( x, y ) ; // no work after recursive call,
just return the value of call
```

The following is NOT tail-recursive.

```
   return rec_func( x, y ) + 3 ; // work after recursive call,
add 3 to result
```

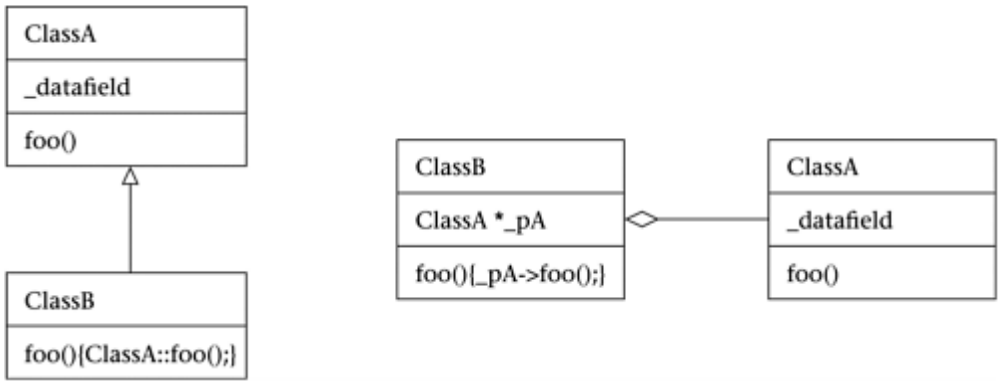
because once **rec_func** is done, it must add 3, and then return that value. Hence, additional work. Hence, not tail-recursive.

It's common, in tail-recursive functions, to have one of the parameters be pass-by-reference (though this is not necessary), which is used to accumulate the answer. Once the base case is reached, you simply return the parameter value, which has the answer. Thus, tail-recursive functions often require an additional parameter, where non tail-recursive functions do not.

Short note on delegation ?

We say that ClassB is composed with ClassA if ClassB has a ClassA or ClassA* member; for short we can say ClassB has a ClassA. And, as before, ClassB inherits from ClassA if ClassB is derived from ClassA as a child class; for short we say ClassB is a ClassA.

As it turns out, you can always replace an inheritance relationship by a composition relationship as indicated in Figure 4.7. If ClassB has a ClassA member object *_pA, then (a) a ClassB object gets a set of ClassA data fields wrapped up inside *_pA and, (b) ClassB can implement the same methods as ClassA simply by passing these method calls off to *_pA. When you pass method calls to a composed object, this is called *delegation*.



Copy constructor ?

The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to:
Initialize one object from another of the same type.
Copy an object to pass it as an argument to a function.
Copy an object to return it from a function.
If a copy constructor is not defined in a class, the compiler itself defines one.If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here:

```
classname (const classname &obj) {
   // body of constructor
}
```

Array of Object ?

C++ Class Objects are several types. They are array of objects, object classes, global objects and local class objects,. In this section of **C++ example programs**, we are going to discuss about 3 source codes. These three C++ tutorial codes are specially designed for array of objects. Beginners like students and professionals can use this sample CPP source code for free of cost.

```
#include<iostream.h>
#include<conio.h>
class c1 // Class definition
{
int m;
int n;
public: //Access specifiers
c1(int o, int p) // Constructor with 2 parameters
{
m=o;
```

```
n=p;
}
int getki()
{
return n;
}
int getih()
{
return m;
}
};

int main() // Main function
{
clrscr();
c1 objj[3]={c1(2,1),c1(2,4),c1(3,5)};
int i;
for(i=0;i<3;i++)
{
cout<<objj[i].getih(); // Output display
cout<< ",";
cout<<objj[i].getki()<<"\n";
}
getch();
return 0;
}
```

Self referential classes ?

The definition of a class may refer to itself. One situation in which this occurs is when a class' operation has an object as a parameter where that object is of that same class as the once containing the operation. Examples of where is occurs are the following:

a Location object is to decide if it has the same screen coordinates as another Location object,

- a Shape object is to decide if it has the same height and width as another Shape object, or
- a File object is to copy itself to or from another File.

In each case the operation needs as its parameter an object in the same class as the one containing the operation.
A second situation in which a class may refer to iself occurs when a method of a class returns an instance of that class as a result. Some examples of methods that return objects in their own class are the following:

- a Shape object returns a new Shape object each of whose dimensions are some percentage less or more than the size of the original Shape object,

- a Location object returns a new Location object that is horizontally or vertically offset from the original Location object, or
- a File object returns a new File object that represents a temporary copy of itself.

In these examples the methods in the Shape, Location or File classes return another object in their same class.
The File class is extended to add a method to perform the copying operations described above. The extended definition is:

```
  class File {                                  // Version 2
    private:
                        // encapsulated implementation goes here
    public:

           File(char* fileName);                // represents file
with given name
           File();                              // unknown, as
yet, file
    char* Name();                               // reply name of
file
    int   Exists();                             // does file
Exist?
    void  View();                               // scrollable view
window
    void  Edit(char* editor);                   // edit file using
"editor"
    void  Delete();                             // delete file
    void  CopyTo(File& other);                  // copy me to
other
    void  CopyFrom(File& other);                // copy other to
me
         ~File();                               // free name
};
```

Write note on 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.
The keyword `this` identifies a special type of pointer. Suppose that you create an object named `x` of class `A`, and class `A` has a nonstatic member function`f()`. If you call the function `x.f()`, the keyword `this` in the body of `f()` stores the address of `x`. You cannot declare the `this` pointer or make assignments to it.

A static member function does not have a `this` pointer.

The type of the `this` pointer for a member function of a class type `X`, is `X* const`. If the member function is declared with the **const** qualifier, the type of the`this` pointer for that member function for class `X`, is `const X* const`.

A `const this` pointer can by used only with `const` member functions. Data members of the class will be constant within that function. The function is still able to change the value, but requires a `const_cast` to do so:

```
void foo::p() const{
member = 1;                          // illegal
const_cast <int&> (member) = 1;    // a bad practice but
legal
}
```


Write A Programs :

Write a program to print even number.

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

int main()
{
int count;
int x = 0;
while (x!=10)//(x!=12)
//I used 10 instead of 12 and it works fine but it
//needs to use 12 instead 10
```

```
{
x = x + 2;
cout << x << "\n""\n";
count++;
cout << endl;
}
return 0;
}
```

Write a program swap a value of two variable by call byrefrance method

```c
#include <stdio.h>

int main()
{
   int x, y, temp;

   printf("Enter the value of x and y\n");
   scanf("%d%d", &x, &y);

   printf("Before Swapping\nx = %d\ny = %d\n",x,y);

   temp = x;
   x    = y;
   y    = temp;

   printf("After Swapping\nx = %d\ny = %d\n",x,y);

   return 0;
}
```

Write a program to greatest and smallest no.in array of 10 element

```
#include <stdio.h>
#include <limits.h> /* For INT_MAX */

/* Function to print first smallest and second smallest elements */
void print2Smallest(int arr[], int arr_size)
{
   int i, first, second;

   /* There should be atleast two elements */
   if (arr_size < 2)
   {
      printf(" Invalid Input ");
```

```
        return;
    }

    first = second = INT_MAX;
    for (i = 0; i < arr_size ; i ++)
    {
        /* If current element is smaller than first then update both
          first and second */
        if (arr[i] < first)
        {
            second = first;
            first = arr[i];
        }

        /* If arr[i] is in between first and second then update second  */
        else if (arr[i] < second && arr[i] != first)
            second = arr[i];
    }
    if (second == INT_MAX)
        printf("There is no second smallest element\n");
    else
        printf("The smallest element is %d and second Smallest element is %d\n",
               first, second);
}
 /* Driver program to test above function */

int main()

{

    int arr[] = {12, 13, 1, 10, 34, 1};

    int n = sizeof(arr)/sizeof(arr[0]);

    print2Smallest(arr, n);

    return 0;

}
```

Operator overloading Program

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

```
class temp
{
   private:
      int count;
   public:
       temp():count(5){  }
       void operator ++() {
        count=count+1;
       }
       void Display() { cout<<"Count: "<<count; }
};
int main()
{
    temp t;
    ++t;         /* operator function void operator ++() is
called */
    t.Display();
    return 0;
}
```

Fibonacci Series up to 10 items

```
#include <iostream>

using namespace std;

int main()
{
    int num1 = 0;
    int num2 = 1;
    int num_temp;
    int num_next = 1;
```

```
    int n;
    cin >> n;
    for (int i = 0; i < n; i++){
        cout << num_next << "  ";
        num_next = num1 + num2;
        num1 = num2;
        num_temp = num2;
        num2 = num_next - num1;
        num1 = num_temp;
    }
    return 0;
}
```

Write a program to demonstrate multilevel and multiple inheritance

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

class A
{
    public:
      void display()
      {
          cout<<"Base class content.";
      }
};

class B : public A
{

};

class C : public B
{
```

```
};

int main()
{
    C c;
    c.display();
    return 0;
}
```

Write a Program for matrix multiplication

```
#include<iostream>
using namespace std;
int main()
{
int l,m,z,n;
int matrixA[10][10];
int matrixB[10][10];
int matrixC[10][10];

cout<<"enter the dimension of the first matrix"<<endl;
cin>>l>>m;
cout<<"enter the dimension of the second matrix"<<endl;
cin>>z>>n;
if(m!=z||z!=m){
cout<<"error in the multiblication enter new dimensions"<<endl;
cout<<"enter the dimension of the first matrix"<<endl;
cin>>l>>m;
cout<<"enter the dimension of the second matrix"<<endl;
cin>>z>>n;
}

else{
cout<<"enter the first matrix"<<endl;
for(int i=0;i<l;i++){
for(int j=0;j<m;j++){
     cin>>matrixA[i][j];
     }
     }
cout<<"enter the second matrix"<<endl;
for(int i=0;i<z;i++){
for(int j=0;j<n;j++){
```

```
    cin>>matrixB[i][j];
}
}
for(int i=0;i<l;i++){
for(int j=0;j<n;j++){
        matrixC[i][j]=0;
        for(int k=0;k<m;k++){
matrixC[i][j]=matrixC[i][j]+(matrixA[i][k] * matrixB[k][j]);
}
}
}

cout<<"your matrix is"<<endl;
for(int i=0;i<l;i++){
for(int j=0;j<n;j++){
cout<<matrixC[i][j]<<" ";
}
cout<<endl;
}
}
//system("pause");
return 0;
}
```

Write a program to show recursive function

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

void numberFunction(int i) {
  cout << "The number is: " << i << endl;
}

int main() {

for(int i=0; i<10; i++) {
  numberFunction(i);
}

return 0;
}
```

Write a program to demonstrate string function

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

```
  void main()
{
        char month[15];
        clrscr();
        cout<<"Enter the String:";
        cin.get(month,15);
        cout<<"The String Entered is "<< month;
        getch();
}
```

Write a program to demonstrate hierarchical inheritance

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

class A  //Base Class
{
   public:
   int a,b;
   void getnumber()
   {
   cout<<"\n\nEnter Number :::\t";
   cin>>a;
   }

};

class B : public A  //Derived Class 1
{
   public:
   void square()
   {
   getnumber();  //Call Base class property
   cout<<"\n\n\tSquare of the number :::\t"<<(a*a);
   cout<<"\n\n\t--------------------------------------------------";
   }
};

class C :public A //Derived Class 2
```

```
{
   public:
   void cube()
   {
   getnumber(); //Call Base class property
   cout<<"\n\n\tCube of the number :::\t"<<(a*a*a);
   cout<<"\n\n\t--------------------------------------------------";
   }
};

int main()
{
clrscr();

B b1;       //b1 is object of Derived class 1
b1.square();  //call member function of class B
C c1;       //c1 is object of Derived class 2
c1.cube();    //call member function of class C

getch();
}
```

Write a program to demonstrate if _else

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

using namespace std;

int main()
{

      // Variable Declaration
      int a;

      // Get Input Value
      cout<<"Enter the Number :";
      cin>>a;

      //If Condition Check
      if(a > 10)
```

```
    {
       // Block For Condition Success
       cout<<a<<" Is Greater than 10";
    }
    else
    {
      // Block For Condition Fail
      cout<<a<<" Is Less than/Equal 10";
    }

    // Wait For Output Screen
    getch();
    return 0;
}
```

Write a program demonstrate copy constructor

```
#include<iostream.h>
#include<conio.h>
class copy
{
              int var,fact;
              public:

                copy(int temp)
                {
                 var = temp;
                }

                double calculate()
                {
                            fact=1;
                            for(int i=1;i<=var;i++)
                            {
                            fact = fact * i;
                            }
```

```
                        return fact;

                }
};
void main()
{
    clrscr();
    int n;
    cout<<"\n\tEnter the Number : ";
    cin>>n;
    copy obj(n);
    copy cpy=obj;
    cout<<"\n\t"<<n<<" Factorial is:"<<obj.calculate();
    cout<<"\n\t"<<n<<" Factorial is:"<<cpy.calculate();
    getch();
}
```

Write a program In_ line function

```
#include <iostream>

using namespace std;

inline int Max(int x, int y)
{
   return (x > y)? x : y;
}

// Main function for the program
int main( )
{

   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) << endl;
   return 0;
}
```

Net references

http://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm
http://msdn.microsoft.com
http://www.cplusplus.com/doc/tutorial/functions/
http://www.rapidprogramming.com
http://en.wikipedia.org
http://www.c4learn.com/cplusplus/cpp-program-structure/
http://publib.boulder.ibm.com
http://www.cplusplus.com
in.answers.yahoo.com
http://www.cpptutor.com
http://www.programiz.com
http://www.programmerinterview.com
http://objectorientedcoldfusion.org
http://www.cppforschool.com
http://www.techopedia.com
www.academia.edu
http://proanswers.org
http://kvsecontents.in/flow-control-statements
http://www.smartclass.co
http://cplus.about.com
http://www.cs.umd.edu
http://etutorials.org
http://people.cs.vt.edu
And many other sites

Feedback :
Chinmayb07@hotmail.co.uk