Why is C++ called object oriented programming/OOP language?
Object Oriented Programming

C++ is called object oriented programming (OOP) language because C++ language views a problem in terms of objects involved rather than the procedure for doing it.

Objects : An object is an identifiable entity with some characteristics and behaviors.

While programming using object oriented approach, the characteristics of an object are represented by its data and its behavior is represented by its functions/operators associated. Therefore, in object oriented programming object represent an entity that can store data and has its interface through functions.

Advantages : Advantages of OOP Programming are that it makes the program less complex, enhances readability. Components are reusable and extendible. Object oriented programming is always good for writing large business logics and large applications or games. Several components of a large program can be easily extended by introducing new classes or introducing new function/operators. OOPs is also very much desired for maintenance and long term support.

Disadvantages : It may not be well suit for small functional problems like adding two numbers or calculating factorial. All the member functions of an object may not be used thus it introduce code overhead.

What are the differentiate between C and C++?
C is a structured programming language.

C++: object oriented programming language.

Programmers use functions/procedures to deal with larger program in C language. In C++ programmers construct a class and related member functions to deal with large class functionality.

C++ can have member functions of structures/classes. C does not support modularization of member functions.

C++ can hide/abstract member variables/functions by private or protected keyword. In C language structure members can not be hidden from outside world. C never support abstraction.

C++ has function and operator overloading/polymorphism feature. C language does not have polymorphism features.

C also does not support inheritance.

What are the advantages/benefits of oop and demerits/disadvantages of oops?

Merits of oops

1. oop provides a programming approach which nearer to the real world problems.
2. oop protects the data from the illegal access. Data is a crucial element of programming.
3. oop divides the whole problem in the form of calls and object.
4. Inheritance gives the reusability of code that keeps limitation the length of the code and that eliminates the redundancy of code.
5. Multiple instances of the object can be used in program without any interference.
6. Actual problem can designed on the basis of objects.
7. Object oriented system can be easily upgraded from small to large system without any change in the internal architecture of the software.
8. Object oriented enables the use of same property in the different way by defining external definition of same internal definition.
9. Message passing gives a better communication technique among the objects.
10. Software complexity can be easily managed.

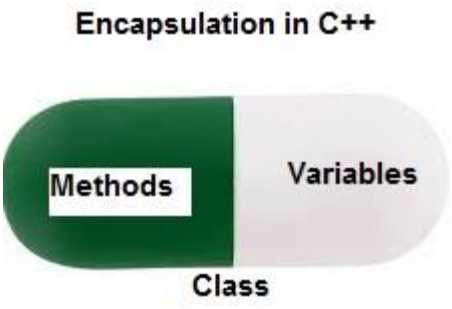
Demerits of oops:

1. Oops puts compiler overhead (that is extra processing time of CPU is required).
2. Object persistency is possible to manage using file handling and data base.
3. Re- orientation of software developer to object oriented thinking requires effort, time mastery over software engineering and programming methodologies.
4. Code reusability is not much easy to achieve.
5. The message passing between many objects in a complex application can be difficult to trace and debug.
6. Benefits only in long run while managing large software projects, at least moderately large one.

What is encapsulation?

encapsulation is the wrapping up of data and variables (that work on the data) into a single unit (called class). The only way to access the data is provided by the functions (that are combined along with the data). These functions are called member functions in C++. If you want to read a data item in an object (an instance of the class), you call a member function in the object. It will read the item and return the value to you. The data is hidden and so is safe from accidental alteration.

Encapsulation is a way of implementing data abstraction. Encapsulation hides the details of the implementation of an object.




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

class Encapsulation
{
   private:
      // data hidden from outside world
      int x;

   public:
      // function to set value of
      // variable x
      void set(int a)
      {
         x =a;
      }

      // function to return value of
      // variable x
      int get()
      {
         return x;
      }};

// main function
int main()
{
   Encapsulation obj;

   obj.set(5);

   cout<<obj.get();
   return 0;
}
```

What are the benefits of encapsulation?

1.protection of data member of object from accidental changes.
2. make it possible to modify the object's implementation without modifying the application that uses it.
3. localize design decisions that are likely to change so as to go to one place for making the changes.

What is a constructor?

A constructor is a class function with the same name as the class itself. It cannot have a return type and may accept parameters. It is a callback or event for the compiler to call the constructor after memory allocation of the object is complete via new operator. Responsibility of constructor function is to set initial values to each object members to some default values before the object can be used.

```
#include<string.h>
#include<iostream.h>
class std12_student
{
private:
  int age;
  char name[20];

public:
  std12_student()
  {
    age = 18;
    strcpy(name, "");
  }
  int get_age()
  {
    return age;
  }
  char* get_name()
  {
    return name;
  }
};
int main(int argc, char* argv[])
{
  std12_student s;
  cout << "Age = " << s.get_age();//Default age = 16
  return 0;
}
```

- Constructor initializes the default name as blank and age of the student as 16.

What is overloaded constructor?

An overloaded constructor is a constructor which takes arguments. It is also called parameterized constructor. Example: Here is an example of student class of standard 12. Default Constructor initializes the default name as blank and age of the student as 16. Overloaded constructor has provisions to give name and age as arguments during construction.

```
#include<string.h>
#include<iostream.h>
class std12_student
{
private:
  int age;
  char name[20];

public:
  /*Default Constructor*/
  std12_student()
  {
    age = 18;
    strcpy(name, "");
  }
  /*Overloaded Constructor 1*/
  std12_student(char *student_name)
  {
    age = 18;
    strcpy(name, student_name);
  }
  /*Overloaded Constructor 2*/
  std12_student(char *student_name, int student_age)
  {
    age = student_age;
    strcpy(name, student_name);
  }
  int get_age()
  {
    return age;
  }
  char* get_name()
  {
    return name;
  }
};
int main(int argc, char* argv[])
{
```

```
 std12_student s("ABC", 17);
 cout << "Name : " << s.get_name() << "Age = " << s.get_age();
 return 0;
}
```

- Default Constructor initializes the default name as blank and age of the student as 16. Overloaded constructor has provisions to give name and age as arguments during construction.

What is a destructor?

A destructor is a special function member of a class that is automatically called when an object about to be deleted by delete operator. It always has the same name as the class preceded by a 'tilt' ~ symbol and has no return type. It is used to free allocated memory used by an object. It takes no arguments.

Example:

```
#include<string.h>
#include<iostream.h>
class student
{
private:
  char *name;

public:
  student()
  {
    name = (char *)new char[20];
    if(name)
    {
      strcpy(name, "");
    }
    else
    {
      name = NULL;
    }
  }
  char* get_name()
  {
    return name;
  }
  void set_name(char *student_name)
  {
    if(name && student_name)
    {
```

```
      strcpy(name, student_name);
    }
    return;
  }
};
int main(int argc, char* argv[])
{
  student * s_ptr;
  s_ptr = new student();
  if(s_ptr)
  {
    s_ptr->set_name("ABC");
    cout << "Student Name : " << s_ptr->get_name();
    delete s_ptr;
  }
  else
  {
    cout << "Memory allocation error";
  }
  return 0;
}
```

Can a destructor be overloaded?

A destructor can never be overloaded. An overloaded destructor would mean that the destructor has taken arguments. Since a destructor does not take arguments, it can never be overloaded.

What do you mean by function overloading?

If more than one function share same function name it is known as function overloading.
Benefits of function overloading: when a programmer has to make a serious program the size of program is overwhelming and the program may use many functions. As the number of function starts growing it become difficult to remember function name.

Even functions doing similar jobs have to be given different function name in c. It gives headache to programmer to remember name of so many functions. C++ allowed keeping name of function same; for function doing similar job provided:
1, if no. of arguments is same then data type must be different.
2, no. of arguments are different then data type can be different or same
Pitfall of function overloading: it is a bad programming idea to create overloaded functions that perform different types of actions therefore function with the same name should have the same general purpose.

What is a copy constructor?

In addition to providing a default constructor and a destructor, the compiler also provides a default copy constructor which is called each time a copy of an object is made. When a program passes an object by value, either into the function or as a function return value, a temporary copy of the object is made. This work is done by the copy constructor.

All copy constructors take one argument or parameter which is the reference to an object of the same class. The default copy constructor copies each data member from the object passed as a parameter to the data member of the new object.

Example:

```
#include<string.h>
#include<iostream.h>
class number
{
private:
  int n;
  public:
  number(int d)
  {
    n=d;
  }
  number(number &a)
  {
    n = a.n;
    cout << "The copy constructor:";
  }
  void display()
  {
    cout << "The number = " << n;
  }
};
int main (int argc, char *argv[])
{
  number n1(10);
  number n2(n1);/*copy constructor called*/
  cout << "object n1:";
  n1.display();
  cout << "object n2:";
  n2.display();
}
```

What is shallow copy?

Shallow Copy:
Default copy constructor of compiler copies all the member variables from source to destination object. This is called shallow copy constructor.

Example:

```
#include<string.h>
#include<iostream.h>
class C
{
 public:
 int var1;
};
void main(int argc, char *argv[])
{
 C c1;
 c1.var1 = 1;
 C c2(c1);
 cout << "value of c2.val " << c2.val;
}
```

What is deep copy constructor?

Overriding default copy constructor with a constructor which copies all the members including all dynamically allocated members is called deep copy constructor.

Differentiate between deep copy constructor and shallow copy constructor?

Deep copy vs Shallow copy:

1. Deep Copy copies all the dynamically allocated members properly to destination.
2. Deep copy is safe and prevents memory corruptions or memory overwriting.
3. When dealing with string, list, vectors, or any other dynamically allocated members deep copy is mandatory.

What is the design of singleton class?

Singleton class does not support object creation/deleting using new and operator. Thus the constructor and destructor have to be private so that it can prevent user from doing so.

In most of the design singleton class provides a interface function like GetInstance() to return an instance to the outside world and also a Release() function to inform the class server that client has finished using this object.

What is the default access modifier for class members and member functions in a class?

In a class, the default access modifier for class member and member functions is private.

Private members are the class members (data members and member functions) that are hidden from the outside world. The private members implement the concept of OOP called data hiding.

What is the default access modifier for structure members and member functions in a structure?

In a class, the default access modifier for class member and member functions is public.

Public members are the class members (data members and member functions) that are open to the outside world. The public function members implement the concept of interfaces in object oriented programming.

What is abstraction or data hiding?

Abstraction or Data hiding refers to the act of representing essential features without including the background details or explanation.

Example: While driving a car, we only know the essential features of the car like gear handling, steering handling, use of clutch, accelerator, and brakes etc. but we do not get into internal details like wiring, engine/motor works etc.

```
Class car

{

public:

/*Public interfaces*/

int get_gear_count(void);

int turn_left(void);

int turn_right(void);

int go_straight(void);

int set_speed(int kmh);

int get_speed(int kmh);

int do_break(void);

private:

/*Private hidden members*/
```

```
  int steering_direction;

  int engine_speed;

  int break_value;

  int current_gear;

};
```

What is THIS pointer?

he member functions of every object have access to a pointer named this, which points to the object itself. When we call a member function, it comes into existence with the values of this set to the address of the object for which it was called. This pointer is the default hidden and implicit argument of any non-static member function.

Using a “this” pointer any member function can find out the address of the object of which it is a member. It can also be used to access the data in the object it points to.

The following program shows the working of the this pointer.

```
class example

{

  private:

  int i;



  public:

  void setdata(int i)

  {

    this->i = i;

    cout << "my object’s address is: " << this << endl;

  }

  void showdata()
```

```
  {
    cout << "my object's address is: " << this << endl;
    cout << this->i;
  }
};
int main (int argc, char *argv[])
{
  example e1;
  e1.setdata(10);
  e1.showdata();
}
```

What is a static function?

In C language if the keyword static appears before any function declaration, it means that the function will now have a file scope or private scope. That is, the static function can be accessed only in the file that declares it. This means, the function is not known outside its own file. Another file may use the same name for a different function.

However in C++ the meaning of static is not the same. Here static member functions are those functions that can be called without any object creation. Scope resolution operator is used to call a static member function.

Example:

```
class C
{
  public:
  static void static_function(void)
  {
  }
```

```
};

void main(int argc, char *argv[])

{

  C::static_function();/*Object not needed*/

}
```

Why THIS pointer will not be created for a static function?

This pointer is available to each member function as a hidden implicit argument. It is the object address value as a pointer. For static functions there is no object address at all because static function can be called even before creation of any object using scope resolution operator. Even calling with an object pointer will not associate any "this" pointer to it. Thus this pointer is not available in any static function.

What is scope resolution operator?

Scope resolution operator, symbol ::, is an operator in C++ which specifies that the scope of the function is restricted to the particular class.
Use of scope resolution operator:
Return-type Class-name :: function-name (parameter list)
{
function body
}

Example:

```
float calc :: sum(float a, float b)

{

   float sum;

   sum = a + b;

   return sum;

}
```

What is an inline function in C++?

The inline functions are a C++ enhancement designed to speed up programs. The coding of normal functions and inline functions is similar except that inline functions definitions start with the keyword inline. The distinction between normal functions and inline functions is the different compilation process.

Working: After writing any program, it is first compiled to get an executable code, which consists of a set of machine language instructions. When this executable code is executed, the operating system loads these instructions into the computer's memory, so that each instruction has a specific memory location. Thus, each instruction has a particular memory address.

After loading the executable program in the computer memory, these instructions are executed step by step. When the function call instruction is encountered, the program stores the memory address of the instructions immediately following the function call statement, loads the function being called into the memory, copies argument values, jumps to the memory location of the called function, executes the function codes, stores the return value of the function, and then jumps back to the address of the instruction that was saved just before executing the called function.

```
inline void max(int a, int b)
{
  cout << (a > b ? a : b);
}
int main()
{
  int x, y;
  cin >> x >> y;
  max(x, y);
}
```

What is the difference between a private member and a protected member?

Private members are class members that are hidden from the outside world. The private members implement the OOP concept of data hiding. The private member of a class can be used only by the member functions of the class in which it is declared. Private members cannot be inherited by other classes.

Protected members are the members that can only be used by the member functions and friends of the class in which it is declared. The protected members cannot be accessed by non member functions. Protected members can be inherited by other classes.

What is a const function?

If a member function of a class does not alter any data in the class, then this member function may be declared as a constant function using the keyword const.

Consider the following declarations:

int max(int,int) const; void prn(void) const;

Once a function is declared as const, it cannot alter the data values of the class. The complier will generate an error message if such functions try to change the data value.

What is polymorphism? Explain briefly.

The process of representing one Form in multiple forms is known as **Polymorphism**. Here one form represent original form or original method always resides in base class and multiple forms represents overridden method which resides in derived classes.

Polymorphism is derived from 2 greek words: **poly** and morphs. The word "poly" means many and **morphs** means forms. So polymorphism means many forms.

Type of polymorphism

- Compile time polymorphism
- Run time polymorphism

Compile time polymorphism

In C++ programming you can achieve compile time polymorphism in two way, which is given below;

- Method overloading
- Method overriding

Method Overloading in C++

Whenever same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as **method overloading**. In below example method "sum()" is present in Addition class with same name but with different signature or arguments.

Example of Method Overloading in C++

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

class Addition
{
public:
void sum(int a, int b)
{
cout<<a+b;
}
void sum(int a, int b, int c)
{
cout<<a+b+c;
}
};
void main()
{
```

```
clrscr();
Addition obj;
obj.sum(10, 20);
cout<<endl;
obj.sum(10, 20, 30);
}
```

Method Overriding in C++

Define any method in both base class and derived class with same name, same parameters or signature, this concept is known as **method overriding**. In below example same method "show()" is present in both base and derived class with same name and signature.

Example of Method Overriding in C++

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

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

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

What is operator overloading?

The functions of most built-in operators can be redefined in C++. These operators can be redefined or 'overloaded' globally or in a class by class basis. Overloaded operators are implemented as functions and can be class-member or global functions.

The name of an overloaded operator is operatorx, where x is the operator. For example, to overload the addition operator, you define a function called operator+. Similarly, to overload the addition/assignment operator, +=, define a function called operator+=.

Although these operators are usually called implicitly by the compiler when they are encountered in code, they can be invoked explicitly the same way as any member or nonmember function is called.

What is function overloading?

```
double max( double d1, double d2 )
{
  return ( d1 > d2 ) ? d1 : d2;
}
int max( int i1, int i2 )
{
  return ( i1 > i2 ) ? i1 : i2;
}
```

The function max is considered an overloaded function. It can be used in code such as the following:

```
main()
{
  int    i = max( 12, 8 );
  double d = max( 32.9, 17.4 );
  return 0;
}
```

In the first case, where the maximum value of two variables of type int is being requested, the function max(int, int) is called. However, in the second case, the arguments are of type double, so the function max(double, double) is called.

What is overriding?

Base class functionality used to get inherited to child classes. Thus if a function is there in base class and a child has been inherited from it, it calls the base function. If child wants to overwrite the base functionality, it should declare the exact prototype in child class as that of the base. Now onwards compiler will use only the child function when applicable. This mechanism is called overriding.

What is a friend class?

The concept of encapsulation and data hiding indicate that nonmember functions should not be able to access an object's private and protected data. The policy is if you are not a member, you

can’t get it. But there is a certain situation wherein you need to share your private or protected data with nonmembers. ‘Friends’ come here as a rescue.

A friend class is a class whose member functions can access another class’ private and protected members. Ex.:

```
class ABC
{
private:
  int x;
  int y;
public:
  void getvalue(void)
  {
    cout << "Enter the values : ";
    cin >> x >> y;
  }
  friend float avg(ABC A);
};
float avg(ABC A)
{
  return float(A.x + A.y)/2.0;
}
int  main()
{
  ABC obj;
  Obj.getvalue();
  float av;
  av = avg(obj);
  cout << "Average = " << av;
  return 0;
}
```

What is a friend function?

The concept of encapsulation and data hiding indicate that nonmember functions should not be able to access an object’s private and protected data. The policy is if you are not a member, you can’t get it. But there is a certain situation wherein you need to share your private or protected data with nonmembers. ‘Friends’ come here as a rescue.

A friend function is a non-member function that grants access to class’s private and protected members.

Pointers for friend functions:

- A friend function may be declared friend of more than one class.

- It does not have the class scope as it depends on function's original definition and declaration.
- It does not require an object (of the class that declares it a friend) for invoking it. It can be invoked like a normal function.
- Since it is not a member function, it cannot access the members of the class directly and has to use an object name and membership operator (.) with each member name.
- It can be declared anywhere in the class without affecting its meaning.
- A member function of the class operates upon the members of the object used to invoke it, while a friend function operates upon the object passed to it as argument.

What is the size of a class having one or more virtual functions?

The sizeof() class with no virtual function inside it, is equal to the sum of all sizes of the member variable. For a class with one or more virtual functions the sizeof() class is the sizeof() class plus the sizeof(void *). This extra size is for the hidden member called vptr which is a pointer, points to vtable.

Example:

```
class C1
{
  short i;
  void funct();
};
class C2
{
  short i;
  virtual void funct();
};
```

C1: size of class C1 is 2 or sizeof(short).

C2: size of class C2 is sizeof(short) + sizeof(void *) that is 2 + 4 = 6 for 32bit compiler. Because this extra 4 byte is used as a hidden vptr member of the class.

What is early binding and late binding?

Early binding: When a non virtual class member function is called, compiler places the code to call the function by symbol name. Thus the function call jumps to a location that is decided by compile time or link time.

```
class base
{
  void funct1(void);
};
base b;
b.funct1(); //compiler call address of base::function() symbol
```

Late binding: When virtual function call is made through a base-class pointer, the compiler quietly inserts code to fetch the VPTR and look up the function address in the VTABLE, thus calling the right function and this is called late/dynamic binding.

```
class base
{
  virtual void funct1(void) = 0;
};
class derived: public base
{
  void funct1(void){}
};
class derived2: public base
{
  void funct1(void){}
};
derived d;
base * b =  d;
//See how b calls function funct1() of d in three steps
//1) get d::vtable address from b::vptr
//   [value: b::vptr points to d::vtable]
//2) get b::funct1() address from b::vtable entry 0
//   [value: function address= d::vtable[0]]
//3) call pointer entry 0 of vtable 0
//   [ call address pointed by d::vtable[0] ]
b->funct1();
derived2 d2;
base * b =  d2;
//See how b calls function funct1() of d2 in three steps
//1) get d2::vtable address from b::vptr
//   [value: b::vptr points to d::vtable]
//2) get b::funct1() address from b::vtable entry 0
//   [value: function address= d::vtable[0]]
//3) call pointer entry 0 of vtable 0
//   [ call address pointed by d::vtable[0] ]
b->funct1();
```

From the above two function call we are clear that we are able to deal with d::vtable and able to call d::funct1() as well as d2::vtable and d2:funct1() accordingly with a single base pointer and the call binding of individual member function was at run time not compile time.

What is inheritance? Briefly explain.

Inheritance is a mechanism of acquiring the features and behaviors of a class by another class. The class whose members are inherited is called the base class, and the class that inherits those members is called the derived class. Inheritance implements the IS-A relationship.

Advantages

1. Reduce code redundancy.
2. Provides code reusability.
3. Reduces source code size and improves code readability.
4. Code is easy to manage and divided into parent and child classes.
5. Supports code extensibility by overriding the base class functionality within child classes.

Different Types of Inheritance

OOPs supports the six types of inheritance as given below-

Single inheritance: In this inheritance, a derived class is created from a single base class.

```
 //Base Class
class A
{
 public void fooA()
 {
 //TO DO:
 } }

//Derived Class
class B : A
{
 public void fooB()
 {
 //TO DO:
 }
}
```

Multi-level inheritance: In this inheritance, a derived class is created from another derived class.

```
 //Base Class
class A
{
 public void fooA()
 {
 //TO DO:
 }
}
```

```
 //Derived Class
class B : A
{
 public void fooB()
 {
 //TO DO:
 }
}

//Derived Class
class C : B
{
 public void fooC()
 {
 //TO DO:
 }
}
```

Multiple inheritance :

In this inheritance, a derived class is created from more than one base class. This inheritance is not supported by .NET Languages like C#, F# etc.

```
 //Base Class
class A
{
 public void fooA()
 {
 //TO DO:
 }
}
//Base Class
class B
{
 public void fooB()
 {
 //TO DO:
 }
}
//Derived Class
class C : A, B
{
 public void fooC()
 {
 //TO DO:
```

```
 }
}
```

Multipath inheritance : In this inheritance, a derived class is created from another derived classes and the same base class of another derived classes. This inheritance is not supported by .NET Languages like C#, F# etc.

```
 //Base Class
class A
{
 public void fooA()
 {
 //TO DO:
 }
}

//Derived Class
class B : A
{
 public void fooB()
 {
 //TO DO:
 }
}

//Derived Class
class C : A
{
 public void fooC()
 {
 //TO DO:
 }
}

//Derived Class
class D : B, A, C
{
 public void fooD()
 {
 //TO DO:
 }
}
```

Hierarchical inheritance : In this inheritance, more than one derived classes are created from a single base.

```
 //Base Class
class A
{
 public void fooA()

 {
 //TO DO:
 }
}

//Derived Class
class B : A
{
 public void fooB()
 {
 //TO DO:
 }
}

//Derived Class
class C : A
{
 public void fooC()
 {
 //TO DO:
 }
}

//Derived Class
class D : C
{
 public void fooD()
 {
 //TO DO:
 }
}

//Derived Class
class E : C
{
 public void fooE()
 {
 //TO DO:
```

```
 }
}

//Derived Class
class F : B
{
 public void fooF()
 {
 //TO DO:
 }
}

//Derived Class
class G :B
{
 public void fooG()
 {
 //TO DO:
 }
}
```

Hybrid inheritance : This is combination of more than one inheritance. Hence, it may be a combination of Multilevel and Multiple inheritance or Hierarchical and Multilevel inheritance or Hierarchical and Multipath inheritance or Hierarchical, Multilevel and Multiple inheritance.

Since .NET Languages like C#, F# etc. does not support multiple and multipath inheritance. Hence hybrid inheritance with a combination of multiple or multipath inheritance is not supported by .NET Languages.

```
 //Base Class
class A
{
 public void fooA()
 {
 //TO DO:
 }
}

//Base Class
class F
{
 public void fooF()
 {
 //TO DO:
 }
```

```
}

//Derived Class
class B : A, F
{
 public void fooB()
 {
 //TO DO:
 }
}

//Derived Class
class C : A
{
 public void fooC()
 {
 //TO DO:
 }
}

//Derived Class
class D : C
{
 public void fooD()
 {
 //TO DO:
 }
}

//Derived Class
class E : C
{
 public void fooE()
 {
 //TO DO:
 }}
```

What is a virtual base class?

We derive a class 'class3' from 'class1' and 'class2' which in turn have been derived from a base class 'base'. If a member function of 'class3' wants to access data or function in the base class, it hits a deadlock. Since 'class1' and 'class2' are both derived from 'base', each inherits a copy of 'base' and each copy, called sub-object, contains its own copy of 'base's data. Now when 'class3' refers to the data in the 'base' class, it is unclear to the compiler which copy to access and hence reports an error. To get rid of this situation, we make 'class1' and 'class2' as virtual base classes.

Using keyword virtual in the two base classes causes them to share the same sub-object of the base class and then 'class1' and 'class2' are known as virtual base class.

How can I overload global << and >> operators to work with cin,cout,cerr etc?

```
friend ostream &operator << (ostream &apm;s, complex &apm;c);
friend istream &operator >> (istream &apm;s, complex &apm;c);
ostream &operator << (ostream &s, complex &c)
{
  s << "(" << c.real << "," << c.imag << ")";
  return s;
}
```

What is an exception? How is exception handling done in C++?

Exception generally refers to some contradictory or unexpected situation or in short an error that is unexpected.
Exception handling is a nice and transparent way to handle errors in programs. For exception handling in C++, the block code to be written is enclosed in a 'try' block. Within the 'try' block, we can throw any occurring errors with 'throw' command. Immediately after the 'try' block, 'catch' block starts wherein error handling code is placed. Thus we can say that there are three keywords for exception handling in C++.

These are:
Try: A 'try' block is a group of C++ statement enclosed in curly braces {}, which may cause (or throw) an exception.
Catch: A 'catch' block is a group of C++ statement that is used to handle a specific raised exception. 'Catch' block should be laced after each 'Try' block. A 'Catch' block is specified by the following:

- The keyword 'catch'
- A catch expression, which corresponds to a specific type of exception that may be thrown by the 'Try' block.
- A group of statements enclosed within braces{} to handle the exception

Throw: This statement is used to throw an exception. A 'Throw' statement specified with:

- The keyword 'throw'
- An assignment expression; the type of the result of this expression determines which catch bock receives control.

Format:

```
try
{
 //: throw exception;
}
```

```
catch(type exception)
{
  //code to be executed in
  case of exception
}
```

What is namespace?

While writing big programs involving several programmers, things are likely to go out of hand if proper control is not exercised over visibility of these names.

```
//mylib.h

char fun()

void display();

class CMath{...};

//somelib.h

class CMath{...};

void display();
```

If both these header files are included in a program, there would be a clash between the two CMath classes. A solution would be to create long names which have lesser chances of clashing but the programmers are required to type these long names. But C++ provides a better solution through a keyword named namespace. C++ provides a single global name spaces. The global name space can be sub-divided into more manageable pieces using name space feature in C++.

```
//mylib.h
namespace myheader
{
char fun()
void display();
class CMath{...};
}
//somelib.h
namespace somelib
{
class CMath{...};
void display();
}
```

The class names will not clash as they become mylib::CMath and somelib::CMath respectively. Same thing would happen to the function names.

What does the keyword 'explicit' do? Why is it used?

Data conversion from standard type to user-defined type is possible through conversion operator

and he class's constructor. But some conversion may take place which we don't want. To prevent conversion operator from performing unwanted conversion, we avoid declaring it. But we may need constructor for building the object. Through the explicit keyword we can prevent unwanted conversions that may be done using the constructor.

What is the difference between dynamic and static casting?

Dynamic cast gives us the flexibility to check the type of object pointer at runtime.

Example:

```
int main (int argc, char *argv[])
{    base * b;
  myclass m, *mp;
  b = &m;
  if(mp = dynamic_cast<myclass *>(b))
  {    cout << " b is type of myclass";
  }   else
  {
    cout << " b is not a type of myclass";
  } }
```

A static cast is used for conversions that are well defined like:

1. cast less conversions,
2. down conversion,
3. conversions from void pointer,
4. implicit type conversion.

What is template class?

Template function is a type of function which only describes the procedure or statements of the function, i.e. what the function will do, but do not specify the data type of arguments. Thus a template class is a basic skeleton.

```
#include
template
T swap(T &a, T &b)
{
  T temp;
  temp = a;
  a = b;
  b = temp;
}
int main (int argc, char *argv[])
```

```
{
  int i = 10, j = 20;
  swap(i, j);
  cout << "Values i, j " << i << ", " << j;
  float a = 3.14, b = -6.28;
  swap(a, b);
  cout << "Values a, b " << a << ", " << b;
}
```

From the above code, it is clear that if template was not used then individual functions have to be used to carry out swap of every data type used.

Template class: Considering a class queue which adds elements to the rear and removes element from the front of an array.

```
#include
const int MAX = 10;
template
class queue
{
        private: T que[MAX];
                 int top;
        public:
};
```

What is function definition?

Function definition contents function declarator and function body. Function body includes all those statements which will complete the task we want to complete using function.

What is function declaration?

Function declaration specifies return data type of function, name of function and data type of arguments, if any. Function declaration is necessary if function is called first before function definition. No. of arguments/parameters and corresponding data type must match when calling function and when writing function declarator.
Function declaration can be dropped if definition of function appears before function call. Function declaration can be global or local. If function declaration appears inside curly brace of any function body it is known as local declaration in such case the function whose function body contains function declaration can call the user defined function. If function declaration does not fall inside curly brace every function can call the user defined function and function declaration is known as global function declaration.