C++ / OOP

What are the most important differences between C and C++?

- C++ supports references while C doesn't.

- Features like friend functions, function overloading, inheritance, templates, and virtual functions are inherent to C++. These are not available in the C programming language.

- In C, exception handling is taken care of in the traditional if-else style. On the other hand, C++ offers support for exception handling at the language level

- Mainly used input and output in C are scanf() and printf(), respectively. In C++, cin is the standard input stream while cout serves as the standard output stream

- While C is a procedural programming language, C++ provides support for both procedural and object-oriented programming approaches

Is it possible for a C++ program to be compiled without the main() function?

Yes, it is possible. However, as the main() function is essential for the execution of the program, the program will stop after compiling and will not execute

OOP:

Object oriented programming is a system in which programs are considered as a collection of objects.

Class:

Is a representation of a type of object.It is a blueprint that describes the details of an object.

Object:

Is an instance of class.It has its own State, behaviour and identity.

Basic of OOP:

1. Abstraction:

It is used for hiding the internal implementation and displays only the required details to the user. Abstraction can be implemented with help of abstract class or interface.

2. Encapsulation:

Encapsulation (also called information hiding) is the process of keeping the details about how an object is implemented hidden away from users of the object. Instead, users of the object access the object through a public interface. In this way, users are able to use the object without having to understand how it is implemented.

In C++, we implement encapsulation via access specifiers(public, private and protected). Typically, all member variables of the class are made private, and most member functions are made public.

3. Inheritance:

It's a process of creating new classes from existing classes. It means to inherit the proprieties of the parents class by the child class. The parent Class is also called: Base class and the child Class called: Derived class. Inheritance is used for code reusability and to extend the parent class;

4. Polymorphisme:

Polymorphisme provides the ability to use the same name for different purposes. Two types of Polymorphism are used:

Static or Compile time polymorphisme:

The polymorphism is implemented at compile time.

- *Where we use the function overloading.*

  Function Overloading: *When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.*

  Operator Overloading: *C ++ also provides options to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator '+' when placed between integer operands , adds them and when placed between string operands, concatenates them.*

Dynamic or Runtime polymorphisme:

Function overriding is an example of Runtime polymorphism.

- *Where we use the function overriding*

  Function overriding: *occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden .*

  In this case parent and child classes both contain the same function with different definitions.

  ⇒ the call of the function is determined at runtime is known as runtime polymorphism

virtual function:

When we declare a function as virtual in a class, all the sub classes that override this function have their function implementation as virtual by default (whether they mark them virtual or not). Why we declare a function virtual? To let compiler know that the call to this function needs to be resolved at runtime (also known as late binding and dynamic linking) so that the object type is determined and the correct version of the function is called.

Object relationships

Composition:

Composition relationships or unidirectional relationships are part-whole relationships where the part must constitute part of the whole object. the object is responsible for the existence of the parts. Most often, this means the part is created when the object is created, and destroyed when the object is destroyed.

Aggregation:

an aggregation is still a part-whole relationship, where the parts are contained within the whole, and it is a unidirectional relationship. However, unlike a composition, parts can belong to more than one object at a time, and the whole object is not responsible for the existence and lifespan of the parts. When an aggregation is created, the aggregation is not responsible for creating the parts. When an aggregation is destroyed, the aggregation is not responsible for destroying the parts

Association:

In an association, the associated object is otherwise unrelated to the object. Just like an aggregation, the associated object can belong to multiple objects simultaneously, and isn't managed by those objects. However, unlike an aggregation, where the relationship is always unidirectional, in an association, the relationship may be unidirectional or bidirectional

Dependencies:

A dependency occurs when one object invokes another object's functionality in order to accomplish some specific task. This is a weaker relationship than an association, but still, any change to the object being depended upon may break functionality in the (dependent) caller. A dependency is always a unidirectional relationship.

Access Specifiers C++:

There are 3 access specifier used to set the accessibility of class, function, method

- *Public: All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.*
- *Private: The class members declared as private can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.*
- *Protected: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.*

Constructor

A class constructor is a special member function of a class that is executed whenever we create new objects of that class. It will have the exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

A constructor is different from normal functions in following ways:

- *Constructor has same name as the class itself*
- *Constructors don't have return type*
- *A constructor is automatically called when an object is created.*
- *If we do not specify a constructor, C++ compiler generates a default constructor for us (with empty body)*

Types of Constructors:

Default Constructors: Default constructor is the constructor which doesn't take any argument. It has no parameters.

Parameterized Constructors: It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created.When you define the constructor's body, use the parameters to initialize the object.

Delegating constructors:

Constructors are allowed to call other constructors. This process is called delegating constructors (or constructor chaining).

If you have multiple constructors that have the same functionality, use delegating constructors to avoid duplicate code.

Copy Constructors:

A member function that initializes an object using another object of the same class is known as a copy constructor in C++. The Copy Constructor can also be made private. A call to the Copy Constructor can happen in any of the following 4 scenarios, when:

- *The compiler generates a temporary object*
- *An object is constructed or based on some another object of the same class*
- *An object of the class is returned by value*
- *An object of the class is passed (i.e. to a function) by value as an argument*

⇒ The general function prototype for the Copy Constructor is:

```
ClassName (const ClassName &old_obj);
```

Example:

```
Point(int x1, int y1) { x=x1; y=y1;}
Point(const Point &p2) { x=p2.x; y=p2.y; }
```

Destructors in C++

Destructor is a member function which destructs or deletes an object.

A destructor function is called automatically when the object goes out of scope:

- *the function ends*
- *the program ends*
- *a block containing local variables ends*
- *a delete operator is called*

Destructors have the same name as the class preceded by a tilde (~). Destructors don't take any argument and don't return anything.

Can there be more than one destructor in a class?

No, there can only be one destructor in a class with class name preceded by ~, no parameters and no return type.

When do we need to write a user-defined destructor?

If we do not write our own destructor in class, the compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

Can a destructor be virtual?

Yes, In fact, it is always a good idea to make destructors virtual in base class when we have a virtual function.Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor.

Pure virtual destructor?

Pure virtual destructor are legal in standard C++ and one of the most important thing is that if class contains pure virtual destructor it is must to provide a function body for the pure virtual destructor. This seems strange that how a virtual function is pure if it requires a function body? But destructors are always called in the reverse order of the class derivation. That means the derived class destructor will be invoked first & then base class destructor will be called. If definition for the pure virtual destructor is not provided so what function body will be called during object destruction? Therefore compilers & linker enforce the existence of a function body for pure virtual destructors.

Should we make all destructors virtual?

It's easy to say yes, so that way you can later use any class as a base class -- but there's a performance penalty for doing so (a virtual pointer added to every instance of your class). So you have to balance that cost, as well as your intent.

Inheritance types:

C++ supports six types of inheritance as follows:

Single Inheritance: A derived class with only one base class is called single inheritance.

Multilevel Inheritance: A derived class with one base class and that base class is a derived class of another is called multilevel inheritance.

Multiple Inheritance: A derived class with multiple base classes is called multiple inheritance.

Hierarchical Inheritance: Multiple derived classes with the same base class is called hierarchical inheritance.

Hybrid Inheritance: Combination of multiple and hierarchical inheritance is called hybrid inheritance.

Multipath Inheritance: A derived class with two base classes and these two base classes have one common base class is called multipath inheritance.

Stream manipulators

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

Abstract Class VS Interface

Abstract Class	Interface
- Can have an instance method and can have an implementation. - Can extend another class and multiple interfaces. - Can not be instantiated - Members can be declared as public, private or protected - Abstract class referred . - Variable or field can be declared as non-final	- can not have any state or implementation; - it can extend the interface only. - can not be instantiated - Member are public - Variable or field are final.

Overriding virtualization

Consider you have

```
class A
{
   virtual void m();
};

class B : public A
{
   virtual void m();
};
```

When you create instance of class B, you will call B::m in all cases:

```
A *a = new B;
a->m(); /// B::m here
```

So, if you want to call m from A? Easy:

```
A *a = new B;
a->A::m(); /// A::m here
```

In the example above I've overridden the virtual call.

Virtual Function in C++

A virtual function a member function which is declared within a base class and is re-defined(Overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call. They are mainly used to achieve Runtime polymorphism.Functions are declared with a virtual keyword in base class. The resolving of function calls is done at Run-time.

Rules for Virtual Functions

Virtual functions cannot be static and also cannot be a friend function of another class.

Virtual functions should be accessed using pointer or reference of base class type to achieve runtime polymorphism.

The prototype of virtual functions should be the same in base as well as derived class.

They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case a base class version of function is used.

A class may have a virtual destructor but it cannot have a virtual constructor.

Pure Virtual function:

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration.

First, Any class with one or more pure virtual functions becomes an abstract base class, which means that it can not be instantiated.Second, any derived class must define a body for this function, or that derived class will be considered an abstract base class as well. A pure virtual function is useful when we have a function that we want to put in the base class, but only the derived classes know what it should return.

Define an Abstract class in C++?

An abstract class in C++ is referred to as the base class, which has at least one pure virtual function. In such a function, a person cannot instantiate an abstract class. This way an Abstract class a pure virtual function is defined by using a pure specifier which is equal to zero during the declaration of the virtual member function in the class declaration. The code sample can be displayed as follows in example.

Friend Class

Friend Class A friend class can access private and protected members of other classes in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of another class. For example a Linked List class may be allowed to access private members of Node.

Friend Function

Friend Function Like friend class, a friend function can be given a special grant to access private and protected members.

A friend function can be:

a) A method of another class

b) A global function

⇒ Following are some important points about friend functions and classes:

1) Friends should be used only for limited purposes. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.

2) Friendship is not mutual. If a class A is a friend of B, then B doesn't become a friend of A automatically.

3) Friendship is not inherited.

4) The concept of friends is not there in Java.

Interface classes

An interface class is a class that has no member variables, and where all of the functions are pure virtual! In other words, the class is purely a definition, and has no actual implementation. Interfaces are useful when you want to define the functionality that derived classes must implement, but leave the details of how the derived class implements that functionality entirely up to the derived class.

Virtual base classes

To share a base class, simply insert the "virtual" keyword in the inheritance list of the derived class. This creates what is called a virtual base class, which means there is only one base object. The base object is shared between all objects in the inheritance tree and it is only constructed once.

Final specifier

There may be cases where you don't want someone to be able to override a virtual function, or inherit from a class. The final specifier can be used to tell the compiler to enforce this. If the user tries to override a function or class that has been specified as final, the compiler will give a compile error.

Covariant return types

There is one special case in which a derived class virtual function override can have a different return type than the base class and still be considered a matching override. If the return type of a virtual function is a pointer or a reference to a class, override functions can return a pointer or a reference to a derived class. These are called covariant return types.

Binding:

Binding refers to the process that is used to convert identifiers (such as variable and function names) into addresses.

Early Binding (static binding):

Direct function calls can be resolved using a process known as early binding. Early binding (also called static binding) means the compiler (or linker) is able to directly associate the identifier name (such as a function or variable name) with a machine address. Remember that all functions have a unique address.

Late Binding (dynamic binding):

In some programs, it is not possible to know which function will be called until runtime (when the program is run). This is known as late binding (or dynamic binding).One way to get late binding is to use function pointers.The function that a function pointer points to can be called by using the function call operator (()) on the pointer.

```
int add(int x, int y)
{
    return x + y;
}
int main()
{
    // Create a function pointer and make it point to the add function
    int (*pFcn)(int, int) = add;
    std::cout << pFcn(5, 3) << std::endl;
}
```

Object slicing:

Remember that derived has a Base part and a Derived part. When we assign a Derived object to a Base object, only the Base portion of the Derived object is copied. The Derived portion is not. In the example above, base receives a copy of the Base portion of derived, but not the Derived portion. That Derived portion has effectively been "sliced off". Consequently, the assigning of a Derived class object to a Base class object is called object slicing

The virtual table:

*The virtual table is a lookup table of functions used to resolve function calls in a dynamic/late binding manner. This table is simply a static array that the compiler sets up at compile time.Second, the compiler also adds a hidden pointer to the base class, which we will call *__vptr.*

**__vptr is set (automatically) when a class instance is created so that it points to the virtual table for that class.*

Overflow error C++:

Arithmetical error, happen when the result of arithmetical operation been greater than the actual space provided by the systems.

Dynamic casting (Upcasting, Downcasting):

Upcasting: C++ implicitly let you convert a Derived pointer into a Base pointer.

Downcasting: Dynamic casting is for converting base-class pointers into derived-class pointers.

Dynamic_cast vs static_cast:

use static_cast unless you're downcasting, in which case dynamic_cast is usually a better choice. However, you should also consider avoiding casting altogether and just using virtual functions.

Downcasting vs virtual functions

In general, using a virtual function should be preferred over downcasting. However, there are times when downcasting is the better choice:

- When you can not modify the base class to add a virtual function (e.g. because the base class is part of the standard library)

- When you need access to something that is derived-class specific (e.g. an access function that only exists in the derived class)

- When adding a virtual function to your base class doesn't make sense (e.g. there is no appropriate value for the base class to return). Using a pure virtual function may be an option here if you don't need to instantiate the base class.

Can we make Operator << virtual?

No, Because we typically implement operator<< as a friend, and friends aren't considered member functions. there's no way to override a function that lives outside of a class (you can overload non-member functions, but not override them).

The solution:

We set up operator<< as a friend in our base class as usual. But instead of having operator<< do the printing itself, we delegate that responsibility to a normal member function that can be virtualized!

Function templates

In C++, function templates are functions that serve as a pattern for creating other similar functions. The basic idea behind function templates is to create a function without having to specify the exact type(s) of some or all of the variables.

Define a class template

A class template is a name given to the generic class. The use of the keyword template is made for defining a class template.

Exception:

C++ provides us with a mechanism to catch all types of exceptions. This is known as a catch-all handler. A catch-all handler works just like a normal catch block, except that instead of using a specific type to catch, it uses the ellipses operator (…) as the type to catch.

Smart pointer:

A Smart pointer is a composition class that is designed to manage dynamically allocated memory and ensure that memory gets deleted when the smart pointer object goes out of scope.

Move semantics:

Move semantics means the class will transfer ownership of the object rather than making a copy.

In C++11, std::move is a standard library function that serves to convert its argument into an r-value.

std::unique_ptr

std::unique_ptr is the C++11 replacement for std::auto_ptr. It should be used to manage any dynamically allocated object that is not shared by multiple objects.

Delete[] VS Delete:

- *Delete: is used to release a unit of memory.*
- *Delete[]: used to release an array.*

std::shared_ptr

std::shared_ptr is a smart pointer that retains shared ownership of an object through a pointer. Several shared_ptr objects may own the same object. The object is destroyed and its memory deallocated when either of the following happens:

- the last remaining shared_ptr owning the object is destroyed;

- the last remaining shared_ptr owning the object is assigned another pointer via operator= or reset().

The object is destroyed using delete-expression or a custom deleter that is supplied to shared_ptr during construction.

std::weak_ptr

std::weak_ptr is a smart pointer that holds a non-owning ("weak") reference to an object that is managed by std::shared_ptr. It must be converted to std::shared_ptr in order to access the referenced object.

std::weak_ptr models temporary ownership: when an object needs to be accessed only if it exists, and it may be deleted at any time by someone else, std::weak_ptr is used to track the object, and it is converted to std::shared_ptr to assume temporary ownership. If the original std::shared_ptr is destroyed at this time, the object's lifetime is extended until the temporary std::shared_ptr is destroyed as well.

Another use for std::weak_ptr is to break reference cycles formed by objects managed by std::shared_ptr. If such cycle is orphaned (i,e. there are no outside shared pointers into the cycle), the shared_ptr reference counts cannot reach zero and the memory is leaked. To prevent this, one of the pointers in the cycle can be made weak.

Namespace C++:

Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes, giving them namespace scope. This allows organizing the elements of programs into different logical scopes referred to by names.

** Namespace is a feature added in C++ and not present in C.*

** A namespace is a declarative region that provides a scope to the identifiers (names of the types, function, variables)*

** Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.*

** A namespace definition begins with the keyword namespace followed by the namespace name*

Operation permitted on pointer:

Incremented & decremented: Increment means that we increment the pointer by the size of data type to which it points.

Pre-increment pointer: (++ptr): increment the operand by 1. Value of expression become the result value of the incremented.

Post-increment pointer: (ptr++): increment the operand by 1. Value of expression will be the value of the operand prior to the incremented value of the operand.

New() VS malloc():

New(): is a preprocessor while malloc(): is a function.

No memory allocation for new but malloc you have to use sizeof().

New initialize the new memory to zero.

Malloc gives a random value in the newly allocated memory location.

New operator is faster than the malloc function, because operator is faster than function.

Overflow error C++:

Arithmetical error, happen when the result of arithmetical operation been greater than the actual space provided by the systems.

Virtual inheritance:

Facilitates to create one copy of each object even if the object appears more than one time in the hierarchy.

Purpose of delete:

delete used to release the dynamic memory created by new.

:: Scope resolution operator:

:: used to define the member function outside the class.

What is the function of the keyword “Volatile”?

"Volatile" is a function that helps in declaring that the particular variable is volatile and thereby directs the compiler to change the variable externally- this way, the compiler optimization on the variable reference can be avoided.

Define storage class in C++? Name some?

Storage classes in C++ specifically resemble life or even the scope of symbols, including the variables, functions, etc. Some of the storage class names in C++ include mutable, auto, static, extern, register, etc.

Can we have a recursive inline function in C++?

Even though it is possible to call an inline function from within itself in C++, the compiler may not generate the inline code. This is so because the compiler won’t be able to determine the depth of the recursion at the compile time.

Nonetheless, a compiler with a good optimizer is able to inline recursive calls until some depth is fixed at compile-time, and insert non-recursive calls at compile time for the cases when the actual depth exceeds at run time.

Define an Inline Function in C++? Is it possible for the C++ compiler to ignore inlining?

Answer: In order to reduce the function call overhead, C++ offers inline functions. As the name suggests, an inline function is one that is expanded in line when it is called.

As soon as the inline function is called, the whole code of the same gets either inserted or substituted at the particular point of the inline function call. The substitution is completed by the C++ compiler at compile time. Small inline functions might increase program efficiency.

Explain 'this' pointer?

The 'this' pointer is a constant pointer and it holds the memory address of the current object. It passes as a hidden argument to all the nonstatic member function calls. Also, it is available as a local variable within the body of all the nonstatic functions.

As static member functions can be called even without any object, i.e. with the class name, the 'this' pointer is not available for them.

What does a Static member in C++ mean?

Denoted by the static keyword, a static member is allocated storage, in the static storage area, only once during the program lifetime. Some important facts pertaining to the static members are:

- Any static member function can't be virtual

- Static member functions don't have 'this' pointer

- The const, const volatile, and volatile declaration aren't available for static member functions

What is a mutable storage class specifier? How can they be used?

A mutable storage class specifier is applied only on non-static and non-constant member variable of the class. It is used for altering the constant class object's member by declaring it. This can be done by using a storage class specifier.

What are the differences between a shallow copy and a deep copy?

The differences between a shallow copy and a deep copy can be stated as under.

Shallow Copy

It allows memory dumping on a bit by bit basis from one object to another.

It is achieved by using a copy instructor and overloading assignment operator.

Deep Copy

It is used for shallow copy purposes.

It allows the copy field, which is done by field from one object to another.

Can we have a String primitive data type in C++?

No, we cannot have a String Primitive data type in C++. Instead, we can have a class from the Standard Template Library (STL).

Can we use access specifiers to achieve data hiding in C++?

Yes, we can use access specifiers to achieve data hiding in C++. These include Private and Protected.

Explain the significance of vTable and vptr in C++ and how the compiler deals with them

vTable is a table containing function pointers. Every class has a vTable. vptr is a pointer to vTable. Each object has a vptr. In order to maintain and use vptr and vTable, the C++ compiler adds additional code at two places:

In every constructor – This code sets vptr:

Of the object being created

To point to vTable of the class

Code with the polymorphic functional call – At every location where a polymorphic call is made, the compiler inserts code in order to first look for vptr using the base class pointer or reference. The vTable of a derived class can be accessed once the vptr is successfully fetched. Address of the derived class function show() is accessed and called using the vTable.

What is the 'diamond problem' that occurs with multiple inheritance in C++?

Answer: The diamond problem in C++ represents the inability of the programming language to support hybrid inheritance using multiple and hierarchical inheritance.