CS304 Object Oriented Programming Lecture Wise Questions and Answers For Final Term Exam Preparation By Virtualians Social Network

Lecture 22 Short notes :

There's 3 type of inheritance in C++

1 Public
2 Private
3 Protected

Public inheritance implements 'is a kind of' relationship.

In public inheritance , public members of the base class become public members of the derived class whereas private parts of the base class can not be accessed by derived class.

So derived class's interface comprises of 2 parts :1st part is derived class's own public member functions and the 2nd and the public member functions of the base class.

Whenever a derived class's object is to be created, the constructor of the base class would be executed first. First, Base class constructor will initialize the base class object and then derived class constructor will initialize the derived class object. From derived class constructor, a base class constructor can also be called explicitly. Base class Constructor Is always called first in either case.

Lecture 23:

Protected access specifier:

If a member is given 'protected access specification' , such member can not be accessed from outside the class. However, a publicly derived class can access such protected members of the base class.

Derived object is a kind of base object. The base class pointer can point towards an object of derived class and vice versa.

cout << pPtr->GetRollNo(); //Error

because pPtr is a pointer to the person class and not the student class. And in person class, we have just assigned a name and not the RollNo(), which belongs to the student class. So we can not assume that a person is also a student, doctor etc. In other words, pPtr's static type is Person, and in Person class, we have not declared RollNo(). So, Access is governed by static type of the object/identity through which a member is being tried to be accessed. Same rule applies in case of Reference (&) as well.

Static Type: The type that is used to declare a reference or pointer is called its static type.

Q. IS-A and HAS-A Relationship

Ans

IS-A means you can have a class that "is a" something. Like a student "IS-A" person. This is used to describe one object being a subclass of another.

Sometimes there isn't a IS-A relationship between two classes, but a "HAS-A" is more appropriate. Example, a chair "HAS-A" leg. Or several. This is called aggregation, as opposed to inheritance.

Is a = Special kind of e.g. Car is special kind of Vehicle.

Has a = It physically has something, e.g Car has an engine.

1. A House is a Building (inheritance);
2. A House has a Room (composition);
3. A House has an occupant (aggregation).

Q. **Default constructor**

Ans

In Default constructor all the parameters have default values. If no constructors are available for a class, the compiler implicitly creates a default parameterless constructor without a constructor initializer and a null body.

Q. Static Type

Ans

The type that is used to declare a reference or pointer is called its static type e.g.

1. In Person * pPtr = 0;

The static type of **pPtr** is Person * (Person pointer).

2. Student s;

The static type of **s** is Student.

Q.Use of Protected key word:

Ans

The protected keyword specifies access to class members in the *member-list* up to the next access specifier (**public** or private) or the end of the class definition. Class members declared as protected can be used only by the following:

- Member functions of the class that originally declared these members.
- Friends of the class that originally declared these members.
- Classes derived with public or protected access from the class that originally declared these members.
- Direct privately derived classes that also have private access to protected members.

Lec 24:

Lec 24:
If we have to initialize a publicly derived class object to some other object through calling copy constructor, if there's no copy constructor in derived and base class, **the compiler will create them automatically.**
First base class's part is copied and later on the derived one's. If user has written the copy constructor then it is necessary to call the base class's copy constructor from derived class's copy constructor.
Base class's copy constructor will called **from the member initialize of the derived class's copy constructor.**

Q. What is the purpose of following keyword in C++? how it is employed, kindly explain with examples:-

typedef enum{ ..... };

Ans

A typedef in C++ is a declaration. Its purpose is to create new types from existing types; whereas a variable declaration creates new memory locations. Every type has constants. For the "int" type, the constants are 1,2,3,4,5; for "char", 'a','b','c'. When a type has constants that have names, like the colors of the rainbow, that type is called an enumerated type.

Example: Use a Typedef To Create An Enumerated Type

typedef enum {RED, BLUE, GREEN} Color;

Color a,b;

a = RED;
a = RED+BLUE; //NOT ALLOWED in eC

if ((a == BLUE) || (a==b)) cout"great";

Q. Q.1 What is eraty of Operator?

Q.2 Does Operator associative means that how the values are assigned to a variable as:

X = 3;

Ans

The value "3" is assigned to variable "x" so this associativity cannot change in Operator overloading?

Operator left or right associative represents which portion of the statement under the operator will execute either left or right. For example,

cout c1 c2;

is equivalent to

operator(**operator(cout,c1)**,c2);

Because insertion operator is Left to right associative so first left part **cout c1** c2 will be executed and then the next part as opposed to copy assignment operator that will right associative.

Q. **Regarding copy constructure and object assingment, what is main difference between them? If we make copy constructure, one object initilize like other object and if we make assingment then also one object is same like other then where is actuall difference? Please explain in detail with atleast one full example**

Ans

When we assigning one object to another object, then behind the scene copy constructor is invoked, that copies the state of one object to another object.

Example:

```
Student::Student (const Student & obj) { //copy constructor with sallow copy

        rollNo=obj.rollNo;

name=obj.name;

GPA=obj.GPA;

}

int main ( ) {

Student studentA;

Student studentB=studentA; // Assigning studentA object to studentB object

/* Shallow copy: compiler will use copy constructor to assign studentA values to newly created object studentB */
```

}

Lec 25:

Over-riding :
Derived class can over-ride/change or modify or totally make new its behavior from that of the bas class.
To over-ride a function , the derived class simply provides a function with the same signature (prototype) as that of its base class.

```
Class parent{
Public:
Void function1();
Void function2(int);}
Class child:public parent{
Public :
Void function1();};
```

Note that the functionality might be the same but classes have to be separate for both the functions i.e the base class's function the the over ridden one.

A direct Base class is explicitly listed in a derived class's header with a colon ; .
An in- direct base class is not explicitly listed in a derived class's header with a colon ; .

> **Q.** what is difference between public and protected and can we use public where needed instead of protected?
>
> **Answer:** A member (either data member or member function) declared in a protected section of a class can only be accessed by member functions and friends of that class, and by member functions and friends of derived classes
> A member (either data member or member function) declared in a public section of a class can be accessed by anyone
>
> **Q.** Protected data members?
>
> Ans
>
> They will become private data members in case of private inheritance and protected data members of derived class in case of protected inheritance.

Lec 26:

Base initialization :

The child/derived class can only call the constructor of its **direct base class** and can not call the constructor of its **indirect base class.**

```
class GrandParent{
int gpData;
public:
GrandParent() : gpData(0){...}
GrandParent(int i) : gpData(i){...}
void Print() const;
};
class Parent1: public GrandParent{
int pData;
public:
Parent1() : GrandParent(), pData(0) {…} // as parent one is direct base class of child 1 so its valid.
};
class Child1 : public Parent1 {
public:
Child1() : Parent1() {...}
Child1(int i) : GrandParent (i) //Error: Child1 can not call its
indirect base class GrandParent Constructor from its constructor
initialization list.
{...}
void Print() const;
};
```

Protected and Private Inheritance :

CHILD class can over-ride the function of any of its Parent class directly or indirectly.

So, a derived/child class contains the objects of its own plus the objects of all the classes up in the hierarchy.

Note that, an anonymous object can not be called by its own name but only through class reference. i.e. class name.

Protected and Private Inheritance :

In private and protected inheritance, base class's protected and public members cant become the interface of the derived/child class.

Note that, A private Inheritance is used when we want to implement one function in terms of another. i.e. Set in terms of Collection or when we want to model " implemented in terms of" relationship and not is a kind of .

Both Collection and Set are same . There exists one difference though, All elements of Set are distinict i.e. not repeated. Whereas , in case of Collection, repetition is allowed.

Q. How many hierarchical level of Inheritance can be achieved in C++?

Answer: The number of *levels of inheritance* is only limited by resources.

Q. Protected Inheritance ?

Ans

In private and protected inheritance, base class's protected and public members cant become the interface of the derived/child class

Q. Private Inheritance ?

A private Inheritance is used when we want to implement one function in terms of another. i.e. Set in terms of Collection or when we want to model " implemented in terms of" relationship and not is a kind of .

Both Collection and Set are same . There exists one difference though, All elements of Set are distinict i.e. not repeated. Whereas , in case of Collection, repetition is allowed.

Lec 27:

In **Specialization** , derived class is behaviourally incompatible (base class can't always be replaced by the derived class) with the base class.

It implements "implemented in terms of relationship".

In **Private inheritance** only member functions and friend classes / functions of a derived class can convert pointer or reference of a derived object to that of parent object.

```
class Parent{
};
```

```
class Child : private Parent{
};
int main(){
Child cobj;
Parent *pptr = & cobj; //Error
return 0;
}
```

The correct statement here would be

Parent & pPtr = static_cast<Parent &>(*this); // fine

here, we are converting the current object is being converted into parents reference.

private inheritance the derived class that is more than one level down the hierarchy cannot access the member functions of grand parent class as public and protected members functions of derived class become private members of privately derived class for all practical purposes.

If a class D has been derived using protected inheritance from class B (If B is a protected base and D is derived class) then public and protected members of B can be accessed by member functions and friends of class D and classes derived from D.
Regards from Persian !

Q. HOW TO SEPRATE THE INTERFACE AND IMPLEMENTATION SO PLZZ PROVIDE ME A SMALL PROGRAM WHICH HAS SEPRATE INTERFACE AND IMPLEMENTAION.

Answer :

Use following way of development

// SomeClass.cpp file for Implementation

```
#include "SomeClass.h"
#include <iostream>

void SomeClass::SomeFunction()
{
  std::cout  "Hello world\n"; //Implementation hidden from user
}
```

```
// SomeClass.h file for Interface

class SomeClass
{
  public:
    void SomeFunction(); //Interface visible to the user
};
```

Q.1 Is it necessary that we must initialize a pointer?

Q.2 When we delete the buffer pointer using:

delete[]buffer ptr;

Why do we use **"[]"** in this code?

Regards,

Answer:

when we use

1. delete p

, in typical situations only the memory space for p[0] gets deallocated. And when we use

1. delete[] p

p then all the memory space from p[0] to p[9] gets deallocated.

Q. If we use protected inheritance; then the derived class can access the protected and public data members and functions of the base class. In this case the public members and functions of base class can be used in the object of the derived class? am I correct? If not, Kindly explain?

Answer: Yes its true and they will be transformed into the protected access specifier.

Lec 28:

Using **Polymorphism** means that different objects can behave in different ways for the same message consequently, sender of a message does not need to know the exact class of receiver.

A function is declared as **Virtual** by preceeding the function header with keyword "virtual".
Static Binding means that target function for a call is selected at compile time –non virtual.

Dynamic Binding means that target function for a call is selected at run time – virtual.
Never over-ride a non-virtual function of a base class.
Regards from Persian !

Q. Describe the way to declare a template function as a friend of any class.???

Ans

```
template<typename T>

class A {

        friend void function( T );

};
```

Q. what is the difference between virtual inheritance and multiple inheritance. Explain the briefly?

Ans

In Virtual inheritance exactly one copy of base class is created.

Whereas in multiple inheritances more then one copy of base class for each child can be created.

Q. Please find me the definition of "Explicit" and "Implicit"?

Answer: In programming; 'Explicit' means that's user will provide the details without having any confusion in it, while 'Implicit' mean that compiler have provided the required details.

Lec 29:
Concrete classes Implements a concrete concept they can be instantiated they may inherit from an abstract class or another concrete class.

In C++, we can make a class abstract by making its function(s) pure virtual.
Conversely, a class with no pure virtual function is a concrete class (which object can be instantiated)

A function is declared pure virtual by following its header with "= 0". **virtual void draw() = 0;**
A class having pure virtual function(s) becomes abstract:

```
class Shape {
…
public:
virtual void draw() = 0;
}
```

Virtual Destructors
Make the base class destructor virtual as we made Draw method virtual in base class.

Virtual Functions – Usage :
when we want to inherit interface and implementation (Simple virtual functions) mean base class as well as derived class will have implementation.
virtual void draw();
Just inherit interface (Pure Virtual functions) mean only derived classes will will have implementation base may not have implementation.
virtual void draw() = 0;

V Table:
A vTable contains a pointer for each virtual function.

in case of virtual functions, for non-virtual functions, compiler just
generates code to call the function whereas In case of virtual functions, compiler generates code to
- access the object
- access the associated vTable
- call the appropriate function

Regards from Persian !

Q. When we use the type **"string"** for data members eg:

"string" RollNo;

Does **"string"** means that the **"RollNo"** can be in characters or in numeric form?

Answer: Data member “RollNo” will be sequence of characters having a Null character at the end as well.

Q. what is the difference between .cpp and .h file ? and how can i make .h file ?

Answer: .CPP file contains implementation of program i.e. the actual code of the program while .h is header file which only contains the interface for the program.

> **Q.** Please explain the meaning of "Anonymous object of base class " ? thanks
>
> Answer: hen we created child class object the implicit base class object is also created along with it, this object is called anonymous base class.

Lec 30-31 :
Polymorphism always works with pointers of class objects not with actual objects.
Multiple Inheritance :
Derived class can inherit from public base class as well as private and protected base classes

class Mermaid: private Woman, private Fish
// As Mermaid class is privately inherited from Woman and Fish so any class derived from Mermaid class will Not have access to public and protected members of Woman and Fish classes

When using public multiple inheritance, the object of derived classes can replace the objects of all the base classes

In virtual inheritance there is exactly one copy of the anonymous base class object
Virtual Inheritance

Virtual inheritance must be used when necessary
There are situation when programmer would want to use two distinct data members inherited from base class rather than one

Q. what is the difference between Default Constructor and implicit Default Constructor ? Thanks

Answer: Default constructor is mostly referred constructor which can be defined by user as well as compiler while implicit default constructor is always defined by compiler.

Q. Please find me the definition of "Explicit" and "Implicit"?

Ans

In programming; 'Explicit' means that's user will provide the details without having any confusion in it, while 'Implicit' mean that compiler have provided the required details.

Q. When we use the type **"string"** for data members eg:

"string" RollNo;

Does **"string"** means that the **"RollNo"** can be in characters or in numeric form?

Regards,

Ans

Data member "RollNo" will be sequence of characters having a Null character at the end as well.

Please explain the meaning of "Anonymous object of base class " ? thanks

Ans

When we created child class object the implicit base class object is also created along with it, this object is called anonymous base class.

Lec 32:

Generic Programming :

Generic programming refers to programs containing generic abstractions (general code that is same in logic for all data types like printArray function), then we instantiate that generic program abstraction (function, class) for a particular data type, such abstractions can work with many different types of data.

In C++ generic programming is done using templates.

Templates are of two kinds,
a. Function Templates (in case we want to write general function like printArray)
b. Class Templates (in case we want to write general class like Array class)

A function template can be parameterized to operate on different types of data types.

```
template< class T >
void funName( T x );

// OR
template< typename T >
void funName( T x );
// OR
template< class T, class U, … >
```

void funName(T x, U y, …);
Note here T is typename of class name and we use this T instead of data type/s for which we want our function to work as template.

Q. what is the difference between Default Constructor and implicit Default Constructor ? Thanks

Ans

Default constructor is mostly referred constructor which can be defined by user as well as compiler while implicit default constructor is always defined by compiler.

Q **What is Static and Dynamic bindings?**

Ans

Static binding means that target function for a call is selected at compile time

- **Dynamic binding** means that target function for a call is selected at run time

Q. What is Virtual Meethod ?
Plz tell its usage and give me at least one example .
thanks

Ans

A virtual function is a member function that you expect to be redefined in derived classes. 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 expression used to make the function call. For example

Shape* _shape = new Line();

_shape->draw(); // Shape::draw called, if draw() is not virtual because of static type of Shape *

Shape* _shape = new Line();

_shape->draw(); // Line::draw called as draw() is virtual in base Shape class

Lec 33 :

Multiple Type Arguments :

the concept of templates can be used here as well to write general function to convert one type into another, in this case we will need two type arguments as shown below,

```
template< typename T, typename U >
T my_cast( U u ) {
return (T)u; // U type will be converted to T type and will be returned
}
int main() {
double d = 10.5674;
int j = my_cast( d ); //Error
int i = my_cast< int >( d );
// need to explicity mention about type of T (int in this case) as it is used only
for // return type not as parameter
return 0;
}
```

Function templates are used when we want to have exactly identical operations on different data types in case of function templates we can not change implementation from data type to data type however we can specialize implementation for a particular data type.

Q. How can i implement member functions like isPlotAllotted((),isPlotCorner(),isDeveloped(),isOccupied() with only data member PlotNo.Please guid me.Thanks

Ans

For functions such as, isOccupied(),isCornerPlot (),isDeveloped () and isPlotAllotted (); you do not need any specific data members. You can implement them in a simple way. You should ask user to give required information as input in respective method. On the basis of given information provided by the user, function will return true or false. For example, in isOccupied() function, prompt user to give input in the from of yes or no. If user enter yes, then this function will return true otherwise false.

Q. can we pass a constructor to a function.....

Ans :

No; we can't pass constructor to a function. A constructor resemble an instance method, but it differs from a method in a sense that it never has an explicit return-type and it is not been inherited. But a constructor can be called in derived class initialization list

Q. if we create a constructor inside a class then why from the main programe we are not allowed to call default c++ constructor.....if we do not create constructor inside a class then c++ default contructor call....

Ans :

Default constructor are used when no constructor are provided. You can overload a default constructor or create a constructor with arguments.

Constructor are used for the specific purpose of creating and initialization of an object. And they differ from other methods in sense that you can't make a direct call to a class constructor like other methods as constructors are only being called when objects are created.

Lec 34 :Generic Algorithms :

We want to provide such implementation that is independent of data structures also for example in case of printing of values we want printing of both single values and arrays, this can be achieved using *Generic Programming* in which a function work for all types of containers
Example : Find function that tries to find an integer value in an integer array

```
const int* find( const int* array, int _size, int x ) {
const int* p = array;
for (int i = 0; i < _size; i++) {if ( *p == x )
return p;
p++;}
return 0;
}
```

```
template< typename P, typename T >
P find( P start, P beyond, const T& x ) {
while ( start != beyond && *start != x )
start++;
return start;
}
```

Note : You can look into the complete procedure of converting the first code to the templates from handouts, here I have just shown you the final form of how the template would look like !

```
int main() {
int iArray[5]; iArray[0] = 15; iArray[1] = 7; iArray[2] = 987; …i
nt* found; found = find(iArray, iArray + 5, 7);
return 0;}
```

Class Templates : definition :

- template< class T > class Xyz { … };
- template< typename T > class Xyz { … };

Q. Sir what is meant by "Abstract class object cannot be instantiated " ?

Ans

Simply means that you can't have an object (location in the memory) of an abstract class. Abstract classes act as expressions of general concepts from which more specific classes can be derived. You cannot create an object of an abstract class type; however, you can use pointers and references to abstract class types.

Q. What is User *user; mean? 2. what is difference between char* name and string name 3. if varchar password then what is that mean?

Ans

User *user; // pointer for the User class type object.
char* name; // pointer for the char type memory space.
string name; // A name object belonging to string type.

Q. what virtual function actually do irrespective of overriding a function?

Ans

The language supplies some built in types (like int, float), but C++ allows you to create your own types. Object Oriented languages allow you to create new types that are based on some existing type and change the behaviour a little using *inheritance*.

In C++ if you want to create a new type that has methods that can be overridden by another object, you declare those methods virtual.

To answer your questions:
1. This is the only time you use virtual functions, when you want to override a function in a base class. It's a design matter which functions are virtual. You don't apply the virtual keyword at random or as you go along.

2. One of the points of OO is that you do not reimplement code already provided in the base class, you simply reuse it. It's an effort saving thing.

3. You can override almost anything in C++, but it's usually not a good idea as the constraints are there for a reason. A class can declare another class to be a friend, and that class will have unfettered access. you don't want to do this as it breaks the structure of the program and creates a program that is unmaintainable.

Lec 35: Member Templates:

Member functions of a template class implicitly become functions templates; they work for instantiations (int, char, float, double so on…) of that class, however there are some situations where we need explicit template functions in our class taking more template parameters other

than one implicit template parameter (parameter given to this class as parameter while creating its object).

A class or class template can have member functions that are themselves templates

```
template<typename T> class Complex {
        T real, imag;
public:
        // Complex<T>( T r, T im )
        Complex( T r, T im ) :
                real(r), imag(im) {}
        // Complex<T>(const Complex<T>& c)
Complex(const Complex<T>& c) :
                real( c.real ), imag( c.imag ) {}
        …
};
```

Q. what is the major difference among static_cast, dynamic_cast, const_cast and reinterpret_cast ?

Ans :

The four type casting operators in C++ with their main usage is listed in the following table:

Type caster keyword	**Description**
static_cast	To convert non polymorphic types.
const_cast	To add or remove the const-ness or volatile-ness type.

dynamic_cast	To convert polymorphic types.
reinterpret_cast	For type conversion of unrelated types.

Q. Generic programming ?

Ans

It is about generalizing software components so that they can be easily reused in a wide variety of situations. In C++, class and function templates are particularly effective mechanisms for generic programming because they make the generalization possible without sacrificing efficiency.

Q. The memcpy() function ?

Ans

It is already generalized to some extent by the use of void* so that the function can be used to copy arrays of different kinds of data. But what if the data we would like to copy is not in an array? Perhaps it is in a linked list. Can we generalize the notion of copy to any sequence of elements? Looking at the body of memcpy(), the function's **minimal requirements** are that it needs to *traverse* through the sequence using some sort of pointer, *access* elements pointed to, *write* the elements to the destination, and *compare* pointers to know when to stop. The C++ standard library groups requirements such as these into **concepts**, in this case the Input Iterator concept (for region2) and the Output Iterator concept (for region1).

Q. Using the generic copy() function?

Ans

we can now copy elements from any kind of sequence, including a linked list that exports iterators such as std::list.

```
#include <list>
#include <vector>
 #include <iostream>
 int main() {   const int N = 3;   std::vector<int> region1(N);
   std::list<int> region2;
    region2.push_back(1);
  region2.push_back(0);
  region2.push_back(3);
```

```
 std::copy(region2.begin(), region2.end(), region1.begin());
for (int i = 0; i < N; ++i)
std::cout  region1[i]  " ";
std::cout  std::endl; }
```

Lec 36: Member Templates Revisited

We can add member templates for ordinary classes as well, for example following code is adding any instance of Complex class to ComplexSet class, (as we know complex class can be instantiated for int, float or double and ComplexSet class will be collection of Complex class instantiations) as shown below,

Complex Class objects,

Complex<int>

real 8

img 3

Complex<double>

real 5.3286

img 5.3284

Q. Object Generators

Ans

An *object generator* is a function template whose only purpose is to construct a new object out of its arguments. Think of it as a kind of generic constructor. An object generator may be more useful than a plain constructor when the exact type to be generated is difficult or impossible to express and the result of the generator can be passed directly to a function rather than stored in a variable. Most Boost object generators are named with the prefix "make_", after std::make_pair(const T&, const U&).

For example, given:

```
struct widget {    void tweak(int); };
std::vector<widget *> widget_ptrs;
```

Q. When do we make a function as private member of the class?

Ans

When you don't want to share it with others but to utilize some functionality within the class.

Q. The destructor is called in reverse order has it is in the lecture.

If we destroy the object of the child class, the destructor of the child class will be called first and then the destructor of the base class.

I wanted to know that if we destroy an object of the base class ,then what will the destructor of the derived class do when it is called 1st?

Ans

Destructor concept is related with objects not with inheritance of an object. So don't confuse them with each other.

Lec 37: Resolution Order
Consider the following specializations of template Vector class,

```
template< typename T >
class Vector { … };
template< typename T >
class Vector< T* > { … };
template< >
class Vector< char* > { … };
```

As these all are template implementations of class Vector, when compiler have to use this template class having many specializations from partial to complete, compiler searches all these specializations in a particular order called resolution order, so resolution order it is the sequence in which compiler searches for required template specialization and is given below,
a. First of all compiler looks for complete specialization

b. If it can not find any required complete specialization then it searches for some partial specialization
c. In the end it searches for some general template
So in other words we can say that compiler searches template specializations from more specific to more general.

Resolution Order
The code below shows which particular instantiation of template Vector class will be used,

```
int main() {
Vector< char* > strVector;
// Vector< char* > instantiated (complete specialization used)
Vector< int* > iPtrVector;
// Vector< T* > instantiated (partial specialization used)
```

```
Vector< int > intVector;
// Vector< T > instantiated (general specialization used)
return 0;
}
```

Q. Q.1 In the code **const char * GetName() const** ,why do we use 2 constants?

Q.2 Can we use "const" before or after the function as:

Ans

1 One const is for return type of the function and other is for the function itself.
2 No as both have different meanings and use.

Q. . static_cast, dynamic_cast, const_cast and reinterpret_cast are all reserve words? 2. typename in templates is a reserve word like class? Thanks

Ans

1 yes all are reserved words
2 yes a reserved word

Q. In generic programming we declare as :-
template <class T>
or
template<typename T>
What does the word 'class' inside the angle bracket signifies....can I write anything 'xyz' instead of class or it
is only class and typename?

Ans

<class T> here class is a reserved keyword.

Lec 38:

Templates and Friends
Templates or their specializations are also compatible with friendship feature of C++ as they are with inheritance.
We consider the following rules when dealing with templates and friend functions and classes.

38.2.Templates and Friends – Rule 1

When an ordinary function or class is declared as friend of a class template then it becomes friend of each instantiation of that template class.

Consider the code below in which we are declaring a class A as friend of a class B and in this class A, we are accessing private data of class B without any error,

In case of friend classes,

```
class A {
…
};
template< class T >
class B {
int data;
friend A; // declaring A as friend of B
…
};
class A {
void method() {
B< int > ib;
B< char > cb
ib.data = 5; // OK: Accessing private member ‘data’ for class B
instantiation ib cb.data = 6; // OK: Accessing private member ‘data’ for class B
instantiation cb
}
};
```

Templates and Friends – Rule 3

When a friend function / class template takes different ‘type parameters’ from the class template granting friendship, then its each instantiation is a friend of each instantiation of the class template granting friendship. Basically here we are removing restriction imposed by Rule 2, due to the use of same type parameter in class B while declaring function doSomething and class A as its friends as shown below,

```
template< class U >
void doSomething( U );
template< class V >

class A { … };
template< class T >
class B {
int data;
template< class W >
```

```
friend void doSomething( W ); // type name is W
template< class S >
friend class A; // type name is S
};
template< class U >
void doSomething( U u ) {
B< int > ib; // Now it is ok to use B for int in function doSomething
instantiated for // char in main
ib.data = 78;
}
```

Q. Plz define Polymorphism?

Ans

Polymorphism menas many shapes, in polymorphism different objects perform different operation for the same message.

Example:

If a “draw” message sends to “circle” object, it draws “circle”.

If a “draw” message sends to “Triangle” object, it draws “triangle”.

If a “draw” message sends to “rectangle” object, it draws “rectangle”.

So there is same message “draw”, but different objects perform different functionality/ behaviour.

Q. Define Dynamic Dispatch?

Ans

Dynamic dispatch is also called dynamic binding. Dynamic binding is a process to map a particular message for a method at runtime. Dynamic binding is most suitable in a situation, where we can not determined to invoke the method at compile time.

Example:

-

Static binding:

Teacher obj1;

obj1.display(); // Static binding of a display()

// At compile time, it is decided that "Teacher" class "display" method will be invoked
//after running the code.

Dynamic binding:

Suppose there are two classes "Person" and "Student", where "Person" is base class, and "Student" is derived class, "display()" is a method of "Person" class.

Person * obj2 = new Student();

obj2->display(); // Dynamic binding

// Now at run time, it will be decide to call/ invoke "Student" class "display" or "Person"
//class "display" method.

}

Q. Plz explain Base class pointer and Base class destructor?

Ans

Base class pointer means a reference of base class, and base class destructor means destructor of base class.

Example:

```
#include<iostream.h>

#include<conio.h>


class Person {

int id;

char * name;

public:

   Person(){

      id=0;

      name=NULL;

      }

~Person(){


   cout"Base class Destructor"endl;
```

```
    } // Base class destructor
};


class Student: public Person {
int id;
char * name;
public:
    Student(){
        id=0;
        name=NULL;
        }
~Student(){
    cout"Derived class Destructor"endl;
    }
};
int main(){
Person * tech= new Student(); // Here tech is a reference of Base class i.e. Person class
delete tech; // and when we delete tech, that is a reference of Person class, therefore it calls the base class destructor.
system("pause");
}
```

Lec 39:

Templates & Static Members

We have studied in case of general classes that their static members are created and initialized at class scope, meaning that memory is reserved for them when a class is compiled and they are initialized as well at that time, they are not dependent on the creation of any object of a class, all these tings are true in case of static members of template classes as well with different that they are created when a template class is instantiated for each date type, each instantiation of template class has its own copy of static data members, suppose we have template class with static members, compiler will create this template class implementation for different date types as required, and each implementation will have its own copy of static data members.16

Consider the class A with a public static data member and a static member function. (It is not good practice to make any data member public but it is declared as public here to demonstrate that separate copy of this data member will be created for each data type).

```
#include <cstdlib>
#include <iostream>
using namespace std;
template< class T >
class A {
public:
static int data;
static void doSomething( T & );
};
template<class T>
int A<T>::data = 0;
int main() {
A< int > ia;
A< char > ca;
ia.data = 5;
ca.data = 7;
cout "ia.data = " ia.data endl "ca.data = " ca.data;
system("PAUSE");
return 0;
}
```

Generic Algorithms Revisited

We studied the concept of Generic Algorithms before that made our code type independent as well as independent of underlying data structure,

For this we developed step by step the following find algorithm that is so generic that it

```
works for all containers,
template< typename P, typename T >
P find( P start, P beyond, const T& x ) {
while ( start != beyond && *start != x )
++start;
return start;
}
int main() {
int iArray[5];
iArray[0] = 15;
iArray[1] = 7;
iArray[2] = 987;
…i
nt* found;
found = find(iArray, iArray + 5, 7);
return 0;
}
```

Q. Q.1 Does **"parameterized constructor"** and **"overloaded constructor"** means the same?

Q.2 Do we have to write the **"destructor"** ourself or it is called automatically when we destroy any object?

Ans

1. No both have there own different meanings and concepts.
2. Destructor is called automatically.

Q. in lecture no 30, page no 240 of handouts, while defining consturctor of class Employee, we write Employee:: Employee(string & n , double tr):name(n){ taxRate=tr;}(1): sir why we write :name(n) in member initializer list and what is its purpose? (2): what is difference between below 2 statements Employee:: Employee(string & n , double tr):name(n){ taxRate=tr;} or Employee:: Employee(string & n , double tr):name(n),taxRate(tr);

Ans

In the following code segment, we are calling “String” class constructor through attribute “name”, and passing parameter “n” to that constructor, therefore we write it in this fashion.

Employee:: Employee (string & n, double tr):name(n) {

taxRate = tr;

}

We write the taxRate in the body of Employee constructor because we just initializing taxRate with tr, not calling "String" class constructor.

The difference between

Employee:: Employee(string & n , double tr):name(n){ taxRate=tr;}

In the above statement, we are calling String class constructor (one parameter) and passing n as a parameter to String class constructor, while taxRate is just initialized in the body of Employee class.

Employee:: Employee(string & n , double tr):name(n),taxRate(tr);

In the above statement, we are calling String class constructor (two parameter) and passing n and tr as a parameter.

Q.i n lecture no 30, when studying the case study of payroll of employees, why we use string class for the name of employees, and why we can not char array to declare name of the employees?

Ans

You can use the char instead of String, but it difficult to manipulate with characters. While the String class has its own overloaded operator, through which manipulation is done effectively. Therefore we use the String instead of char, and also to implement the relationship given in the object model of String, Employee.

Lec 40: Cursors
A better way is to use cursors
A cursor is a pointer that is declared outside the container / aggregate object

Aggregate object provides methods that help a cursor to traverse the elements

- T* first()
- T* beyond()
- T* next(T*)

In this approach we will add three methods given above in our Vector class that will return pointer to elements of Vector elements instead of whole vector object.

Now our vector class is as follows,

Vector

```
template< class T >
class Vector {
private:
T* ptr;
int size;
public:
Vector( int = 10 );
Vector( const Vector< T >& );
~Vector();
int getSize() const;
const Vector< T >& operator =( const Vector< T >& );
T& operator []( int );
T* first();
T* beyond();
T* next( T* );
};
template< class T >
T* Vector< T >::first() {return ptr;
}
template< class T >
T* Vector< T >::beyond() {
return ( ptr + size );
}
template< class T >
T* Vector< T >::next( T* current )
{
if ( current < (ptr + size) )
return ( current + 1 );
// else
return current;
}
int main() {
Vector< int > iv( 3 );
```

```
iv[0] = 10;
iv[1] = 20;
iv[2] = 30;
int* first = iv.first();
int* beyond = iv.beyond();
int* found = find(first,beyond,20);
return 0;
}
```

Conclusion

The main benefit is that we are using external pointer so we can do now any kind of traversal in any way i.e. as many traversals as we need only we will need one pointer for each traversal, however we can not use cursors in place of pointers for all containers.

40.2.Iterators

We studied cursors previously, cursors were external pointer that we accessing internal data of any container like vector, it is against the principle of data hiding as we can access any container data using cursors so it is not good programming practice to given access in container for the use of cursors (first, next, beyond methods) we have alternate to cursors in the form of Iterators which are that traverse a container without exposing its internal representation. Mean they are not external pointers but internal data member of container. Iterators are for containers exactly like pointers are for ordinary data structures (you can see this line as we have made a mechanism to declare pointers to our containers like we declare pointers to ordinary data types, in case of cursors we were using ordinary data type pointer but in case of Iterators we will use container pointers without exposing their internal details)

Generic Iterators

General Iterator class can point to any container because it is implemented using templates, basically it provides us the functionality to create any container object using templates and operator overloading this has been explained in code below, now we will not directly use methods first , beyond and next of container but we will use them through Iterator class i.e we will create Iterator class objects and will iterate container through these methods, however these objects will not be able to directly access internal elements of container. A generic Iterator works with any kind of container. We need the same set of operations in container class to use Iterators,

- T* first()
- T* beyond()
- T* next(T*)

Q. What is difference b/w Multiple inheritance and Virtual inheritance? Sir, i have checked it in lecture 27, but i did not find solution it. Plz send me solution again. i will be thankful to you.

Ans

In Virtual inheritance exactly one copy of base class is created, where's in multiple inheritances more then one copy of base class for each child can be created.

Q. I did't able to understand how copy constructor's code work Student::Student(const Student &obj){ rollNo=obj.rollNo; --- --- } Q is , we are passing obj reference as argument...how it works??? and whats the logic of obj.rollNo?? when we write Std2=Std1; how data members are copied one from the other...plz explain the logic...

Ans

When we write Std2=Std1; the copy constructor is invoked, and data members of one object is copied to another object.

As in the definition of copy constructor,

Student::Student(const Student & obj){

rollNo=obj.rollNo;

}

In the above code segment, we are assigning the data members of already created object i.e. Std1, to newly created object Std2.

We are passing a reference of object, which points to the memory location of Student class object obj. obj.rollNo means that the rollNo of Std1 object (obj) is accessed(which is already stored in memory), and that value is assigned to the rollNo of object Std2.

So the data member rollNo of Std1 is copied to the rollNo of Std2 object.

Q. Please let me know the function in DEV C++, which can be used for clearing DOS screen. Thanks

Ans

You can use system("cls") for screen clear.

Example:

```
#include<iostream.h>
template < typename T>
T max( T a, T b){
return a>b ? a: b;
}
int main(){

   coutmax(3,7);

   system("cls");

   coutmax(3,7);
   system("pause");
}
```

Lec 41: Standard Template Library:
When we started discussing about templates and generic algorithms, our basic aim was to make standard solutions for different problems (like searching or comparison) that should work for all data types and for all containers.
C++ programmers started working on these concepts from very beginning and gave many standard solutions to these problems these standard solutions after the approval of C++ standardization committee were added to a combined library name as Standard Template Library (STL). STL is designed to operate efficiently across many different applications; we can use solutions from this template library in our programs using different header files.

Standard Template Library
STL consists of three key components
• Containers
• Iterators
• Algorithms
STL Promotes Reuse
STL promotes reuse as we don't need to rewrite the already written standard code for different problems, it saves our development time and cost. Secondly these solutions have been thoroughly tested so there is no change of error due to their use.
41.2.STL Containers
Container is an object that contains a collection of data elements like we have studied before now we will study them in detail.

STL provides three kinds of containers,
1. Sequence Containers
2. Associative Containers
3. Container Adapters

First-class Containers
Sequence and associative containers are collectively referred to as the first-class containers
Container Adapters
A container adapter is a constrained version of some first-class container

Q. Please let me know the function in DEV C++, which can be used for clearing DOS screen. Thanks

Ans

You can use system("cls") for screen clear.

Example:

```
#include<iostream.h>
template < typename T>
T max( T a, T b){
return a>b ? a: b;
}
int main(){

   coutmax(3,7);

   system("cls");

   coutmax(3,7);
   system("pause");
}
```

Q. Dear Sir, Define Template and Template function?

Ans

Templates are the features of C++ that enables classes and functions to operate with generic types. Through templates a class or a function works on many different data types without being rewritten for each one.

Template functions:

Template functions are operated with generic data types. Template function is a function having a template argument (type argument).

-

Example:

-

A simple function:

int max(int a, int b){

return a>b ? a: b;

}

Only takes integer values

A template function:

template < typename T>

T max(T a, T b){

return (a>b ? a: b);

}

Can take any data type i.e. int, float, char etc

Q. plz explain dereference operator. if we dont use this operator in a program then what would be happened?

Ans

It is possible to access the value of variables pointed by the pointer variables using pointer. This is performed by using the Dereference operator in C++ which has the notation *.

The general syntax of the Dereference operator is as follows:

*pointer_variable

In this example, pointer variable denotes the variable defined as pointer. The * placed before the pointer_variable denotes the value pointed by the pointer_variable.

For example:

```
exforsys=100;
test=exforsys;
x=&exforsys;
y=*x;
```

The indirection operator (*) is used in two distinct ways with pointers: declaration and dereference. When a pointer is declared, the star indicates that it is a pointer, not a normal variable. For example,

```
unsigned short * pAge = 0; // make a pointer to an unsigned short
```

When the pointer is dereferenced, the indirection operator indicates that the value at the memory location stored in the pointer is to be accessed, rather than the address itself.

*pAge = 5; // assign 5 to the value at pAge

Lec 42:

Iterators
We have studied about Iterators before; they are used to traverse Containers efficiently without accessing internal details, now we see Iterators provided to us in standard template library, Iterators are for containers like pointers are for ordinary data structures STL Iterators provide pointer operations such as * and ++
42.2.Iterator Categories
We can divide Iterators given to us in STL in the following categories,
a. Input Iterators
b. Output Iterators
c. Forward Iterators
d. Bidirectional Iterators
e. Random-access Iterators
Input Iterators
Using input iterators we can read an element, and we can only move in forward direction one element at a time, these can be used in implementing those algorithms that can be solved in one pass (moving container once in single direction from start to end like find algorithm we studied in last lecture).
Output Iterators
Using output iterators we can read an element, and we can only move in forward direction one element at a time, these can be used in implementing those algorithms that can be solved in one pass (moving container once in single direction from start to end like find algorithm we studied in last lecture).
Forward Iterators
Forward iterators have the capabilities of both input and output Iterators, in addition they can bookmark a position in the container (we can set one position as bookmark while traversing the container this will be more understandable when we will see example below)
Bidirectional Iterators
They have all the capabilities of forward Iterators plus they can be moved in backward direction, as a result they support multi-pass algorithms (algorithms that need more that need to traverse container more than once).
Random Access Iterators
They have all the capabilities of bidirectional Iterators plus they can directly access any element of a container.
42.3.Iterator Summary:
Following diagram shows the capabilities and scope of different iterators you can see

that Random access iterators have all the capabilities and input and output iterators have least capabilities.

Q. how can derived class initialize data members of base class. please explain this line Student::Student(char * _name, char * _mail):Person (_name), major(NULL) Here why we use Person constructor in member initializer list , and other question is major data member of student class cannot be initialized with out this way or method. why we initialized major here. please explain thanks

Ans

Student::Student(char * _name, char * _mail):Person (_name), major(NULL)

The constructor of student class is taking two char *'s as inputs, one is name of student and second is major subject of that student.

In initialization list we are initializing major with null value and we are also calling base class constructor of person class as you know that base class part of an objects is created first and then its derived class part is created. So, if we don't mention explicitly

Person (_name) in derived class constructor, then we can't achieve the real sense on Inheritance in C++.

In Code of student constructor in which are checking maj for NULL if it is not we are creating memory equal to passed value and assigning it to char * major.

Q. there is no visual language like visual basic in our MCs course. Are such languages are not important and without learning them would we be able to make project of course?. what is requirement of market or industry what programming expertise should we have?. thanks

Ans

There are too many programming courses that are certainly help you in developing a small, medium as well as large scale projects and it's as per market.

For example if you want to develop an OO system using Java, C++, Windows (Visual) programming, whether the application is of desktop or web based (Web Based Design and Development), System Programming is also the part of your course, where you can learn low level embedded system programming, You have all these in your hand.

Secondly, in order to develop a system which is according to Software Engineering principles, you will have Software Engineering l and ll (Construction and Management) in a very detailed and precise way.

Thirdly, definitely you need to understand the concepts and implementation of Data Bases in depth which is also provided by VU.

As you know very well that most of the languages used to develop software systems in the market are conceptually almost same, only syntactically they differ. So, my advice you to get grip over defined languages and material provided in your study scheme and I am sure that while you will be in market, will have a lot of with you in hands

Q. Why Program is not compiling?

#include using namespace::std; using std::cout; using std::cin; using std::endl; class CaseSenCmp { public: static int isEqual(char x, char y) { return x == y; } }; class NonCaseSenCmp { public: static int isEqual(char x, char y) { return toupper(x) == toupper(y); } }; template< typename C = CaseSenCmp > int compare(char* str1, char* str2) { for (int i = 0; i < strlen(str1) && i < strlen(str2); i++) if (!C::isEqual (str1[i], str2[i])) return str1[i] - str2[i]; return strlen(str1) - strlen(str2); }; int main() { int i, j; char *x = "hello", *y = "HELLO"; i = compare< CaseSenCmp >(x, y); j = compare< NonCaseSenCmp >(x, y); cout "Case Sensitive: " i; cout "\nNon-Case Sensitive: " j endl; system("pause"); return 0; }
Ans

Giving error on this line:

template< typename C = CaseSenCmp > int compare(char* str1, char* str2)

Default template arguments may not be used in function templates.

It works fine with

template< typename C > int compare(char* str1, char* str2)

Hope it helps you

Lec 43 : Techniques for Error Handling:
Sometimes our program terminates abnormally, sometimes they even crash the system, these errors occur mostly due to incorrect memory access or due to input/output error, sometimes it is our program fault and sometimes it is some external resource error (like network or hard disk).
If we allow these errors to happen we may lose our work for example if a text editor terminates abnormally without allowing us to save our work we will lost our work, so it is important that we add some type of error handling mechanism in our program, we use the following techniques for error handling,
a. Abnormal termination
b. Graceful termination
c. Return the illegal value
d. Return error code from a function
e. Exception handling

Program can be designed in such a way that instead of abnormal termination, that causes the wastage of resources, program performs clean up tasks, mean we add check for expected errors (using if conditions),
Example – Graceful Termination

```
int Quotient (int a, int b ) {
if(b == 0){
cout “Denominator can’t “ “ be zero” endl;
// Do local clean up
exit(1);
}
return a / b;
}
```

Output
Enter two integers
10
10
Quotient of 10 and 10 is 1
Enter two integers

10
0
Denominator can't be zero

Error Handling
- Issues in Error Handling:
- Programmer sometimes has to change the design to incorporate error handling
- Programmer has to check the return type of the function to know whether an error has occurred
- Programmer of calling function can ignore the return value
- The result of the function might contain illegal value, this may cause a system crash later
- Program's Complexity Increases

The error handling code increases the complexity of the code
o Error handling code is mixed with program logic
o The code becomes less readable
o Difficult to modify
The example below shows these concepts,
Example- without error handling

```
int main() {
function1();
function2();
function3();
return 0;
}
```

Q. There is a confusion in understanding about the private members of class during inheritance. My understanding is that all the private members of the base class are private in derived class and their further inheritance to the down level derived class is not possible.... The table at page number 211 shows the member access. I found that all private members of base class are hidden to the derived class instead of private.... Sir kindly elaborate. Thanks in advance, Sir.

Ans

All private data members of Base class in case of Public, Protected or Private Inheritance acts as hidden data members in derived class. Means all said data members can't be accessed directly in derived class. Such private data members can be accessed through defined exposed interface of base class.

Q. the words "template" or "class" both can be used for template functions and classes..... Can we use any other word instead of these twos just like variable name or not?

Ans

We can't use any other word.

Q. the code return x == y ..... is it means if both the x & y are equal then return TRUE otherwise FALSE? Thanks sir.

Ans

It means if condition is true then execute all statments related to this condition, if it is not true then, skip all those statments and move to another condition next to it..

Lec 44: Stack Unwinding

The flow control (the order in which code statements and function calls are made) as a result of throw statement is referred as “stack unwinding”

Stack Unwinding can take place in the following two ways,

1. When we have nested try catch blocks (one try catch block into other try catch block), for example

```
try {
try {
} catch( Exception e) {
}
} catch(exception e){}
```

2. When exception is thrown from nested functions having try catch blocks

```
void function1() {
throw Exception();
}
void function2() {
function1();
}
int main() {
try{
function2();
} catch( Exception ) { }
return 0;
}
```

Catch Handler

- We can modify the code in catch handler to use the exception object to carry information about the cause of error
- The exception object thrown is copied to the object given in the handler
- We pass the exception as reference instead of by value in the catch handler to avoid problem caused by shallow copy

Catch Handler

The object thrown as exception is destroyed when the execution of the catch handler completes

Avoiding too many Catch Handlers

There are two ways to catch more then one object in a single catch handler

- Use inheritance
- Catch every exception

Q. on page 262 code written below. return strcmp(x.pStr, y.pStr)==0 I would like to understand it. Means that the output of strcmp(x.pStr, y.pStr) expression is comparing with zero (0) ??? Please elaborate it,

Ans

return strcmp(x.pStr, y.pStr) == 0;

It's just a comparison between two strings. And after comparison if both are equal then, return "true" (which is ==0 here) , otherwise return false.

Hope it helps you

Q. why we use member intilizer list.what is benefit of using it.

where we use it.

Ans

A proper initialization of data member has a special syntactic construct called a *member initialization list*.

For example:

```
class Person

{

//..

public:

//member initialization list:

 Person(int a, const string & n) : age (a), name (n)

 {}

};
```

A member initialization list is the primary form of initializing data members of an object. In some cases, initialization is optional. However, const members, references and sub objects whose constructors take arguments necessitate a member initialization list.

Hope it helps yo

Q. Why we use Getter Setter function in our program and what is const keyword ?

Ans

Getter and setter functions and const keyword are discussed in video lectures and handouts in detail. Go through them and if you still feel confusion or difficulty in understanding them then inform us to clear it for you.

Lec 45:

Resource Management

- Function acquiring a resource must properly release it
- Throwing an exception can cause resource wastage

Example

```
int function1(){
FILE *fileptr = fopen("filename.txt","w");
...
throw exception();
```

```
...
fclose(fileptr);
return 0;
}
```

In case of exception the call to fclose will be ignored and file will remain opened.

First Attempt

```
int function1(){
try{
FILE *fileptr = fopen("filename.txt","w");
fwrite("Hello World",1,11,fileptr);
...
throw exception();
fclose(fileptr);
} catch(...) {
fclose(fileptr); // adding fclose in catch handler as well
throw;
}
return 0;
}
```

Exception in Constructors

Exception thrown in constructor cause the destructor to be called for any object built as part of object being constructed before exception is thrown

Destructor for partially constructed object is not called

Example

```
class Student{
String FirstName;
String SecondName;
String EmailAddress;
…
}
```

If the constructor of the SecondName throws an exception then the destructor for the First Name will be called.

So generally we can say that in constructor if an exception is thrown than all objects created so far are destroyed, if EmailAddress String object had thrown exception then SecondName and FirstName objects will be destroyed using their destructor. However destructor of Student class itself will not be called in any case as its object was not completely constructed.

Q. if there is any special reason (logic), to hide the search method in assignment requirement file, as without the search method a concrete class (owner), is not possible to build ?

Ans

Owner class doesn't require a search method, as owner in real life has the knowledge of his or her own property. But in our case due to the situation where a search method can be handy due to hard coded property names we have made it pure virtual in the base class. So, it makes overriding this method as a necessity in all the child classes. And now a student can either use and implement search method in the Owner class or just override the search method without any solid implementation (in this case a cout statement will be enough).

Q. What is the philosophy of return statement. I know when we want to return any value to the function, we need to code the value along with return statement. But here is the code of linear search, if i return the value after cout statement, i can see only the location in one out put and if i return the statement before the cout i can see the entire process. Please guide me the **Logic** of return.
Regards
Kashif.

```
int isPresent(){

int array[29] =
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,115,16,17,18,19,20,21,22,23,24,25,26,27,28};//should
be sorted/115 not-sorted.
int val=0;
int high = 29;
int low = 0;
int mid=0 ;
int n = 0;
int k = 0;
int m;
cout<<"Enter The Degit "<<endl;
cin>>val;

for(int i=0;i<= 28;i++){
    array[i];
   if(array[i] == val){
```

```
            return array[i];//returning array[i] in if statement, while if statment is embeded in
for loop.

  }

   cout<<"Count: "<<i+1<<"=>"<<" The linear search result:
"<<array[i+1]<<endl;//belongs to for loop.
      }


       }
```

Ans

Return statement, terminates the execution of a function and returns control to the calling function (or, in the case of the main function, transfers control back to the operating system). Execution resumes in the calling function at the point immediately following the call.

return [expression]

In the case of code snippet you shared, the return statement only executes when if condition becomes true. That is the value you had taken from user is found in the array. Whereas the case concerning cout statement based output is coming with respect to the loop but not with return statement value.

Q. Here is the code, its out put is 2. why?

```
#include<iostream>
using namespace std;
int main(){
 int hole;
 cout<<hole<<endl;
 system("pause");

   }
```

Regards

Ans

The output shown on the screen is garbage value as you did not assign any value to variable hole. You should initialize it with some logically correct value. For "char c", initialized it with some value and then print value stored in variable C such as:

```
char c= 'a';
cout<<c<<endl;
```

Q. please can you define that what is Static Casting?

Ans

Static casting is a conventional casting of one type to another, where there is some meaning to what the cast will do. (Thus, normally you can't cast from a **Foo** to a **Bar**, unless there's a definition for how to do this).

The main kind of static casting we're interested in falls into two categories:

1. Casting from one primitive type to another (e.g., **char** to **int**, **float** to **int**).

2. Casting from a shorter type to a longer type or vice versa (e.g., **short** to **int**, **long** to **int**).

Casting from one type to another is interesting, particularly, casting **float** to **int**. Like the unary minus, the casting operation does NOT change the value of the variable it is casting. Instead, like unary minus, it creates a temporary value which is the casted result.

For example, consider

```
float f = 3.0 ;

int val = static_cast<int>( f ) ;
```

The static cast of **f** to an int does not change the value of **f**. Instead, it creates a temporary **int**.