**Sunbeam Infotech**

**CPP Notes**

**Day 4**

**\*Scope**

It decides area/region/boundry in which we can access the element.

**Types of scope in C++:**

**1. Block scope**

**2. Function scope**

**3. Prototype scope**

**4. Class scope**

**5. Namespace scope**

**6. File scope**

**7. Program scope**

**\* Example Scope**

int num6; **//Program Scope**

static int num5; **//File Scope**

namespace ntest

{

int num4; **//Namespace scope**

class Test

{

int num3; **//Class Scope**

};

}

void sum( int num1, int num2 ); **//Prototype scope**

int main( void )

{

int num1 = 10; **//Function Scope**

while( true )

{

int temp = 0;

}

return 0; **//Block Scope**

}

**\* Macro**

* Symbolic constant is called as macro
* Expanding macro is a job of preprocessor.
* Example:
  + #define SIZE 10
  + #define EOF -1

**\* Friend Function**

If we want to access private members inside derived class

* Either we should use member function(getter/setter).
* Or we should declare a facilitator function as a friend function.
* Or we should declare derived class as a friend inside base class.

**\* OOPS Concepts**

**Major Pillars/Parts/Elements:**

**1. Abstraction**

* If we create object and call member functions on it then it represents abstraction programatically.
* Abstraction describes outer behavior of object.
* Example:

{

Point p;

p.accept\_data();

p.display\_data();

}

**2. Encapsulation**

* It is to achive abstraction and data hiding.
* Binding of data and code together is called encpasulation.
* To achive abstraction we need to provide some implementation. It is called encapsulation.
* By defining class we achive encapsulation
* Encapsulation describes internal behavior of object
* Abstraction focuses on the observable behavior of an object, whereas encapsulation focuses on the implementation that gives rise to this behavior.
* Data hiding is also called data encapsulation.
* Process of declaring data member private is called data hiding.

**3.Modularity**

* Process of developing complex system using small parts is called modularity.
* Main purpose of modularity is to minimize module dependancy.

**4.Hierarchy**

* Its main purpose is to achive reusability.

**Minor Pillars/Parts/Elements**

**1. Typing/ polymorphism**

It is also called as polymorphism.

An ability of object to take multiple forms is called polymorphism.

Types of Polymorphism:

1. Compile time polymoprhism

It is also called as static polymoprhism / Early Binding / False polymoprhism / Weak Typing

We can achive it using:

1. Function Overloading

2. Operator Overloading

3. Template

1. Runtime polymorphism

It is also called as dynamic polymoprhism / Late Binding / True polymoprhism / Strong Typing

We can achive it using:

1. Function Overriding (can be achived using virtual)

**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.**

What is the use of virtual keyword?  
Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object.

**Early Binding**

When we use Base class's pointer to hold Derived class's object, base class pointer or reference will always call the base version of the function.

**Late Binding**

## **Using Virtual Keyword in C++**

We can make base class's methods virtual by using **virtual** keyword while declaring them. Virtual keyword will lead to Late Binding of that method.

On using Virtual keyword with Base class's function, Late Binding takes place and the derived version of function will be called, because base class pointer pointes to Derived class object.

**Points to note for Virtual keyword:**

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

**\* Pure Virtual Function**

**virtual returntype functionname([argument\_list]) =0;**

**\*Abstract Class**

Sometimes implementation of all function cannot be provided in a base class because we don’t know the implementation. Such a class is called abstract class.

**2. Concurrency**

* Process of executing multiple task simultaneously is called concurrency.

**3. Persistence**

* It is the process of maintaing state of object on secondry storage(HDD).
* We can achive it using file handling and database programming.

**\*Association**

* If has-a relationship exist between two types then we should use association.
* Example : Car has-a engine (OR engine is part-of car)
* If object is part-of / component of another object then it is called association.
* Composition and aggregation are specialized form of association.
* **If we declare object of a class as a data member inside another class then it represents association.**

**Association Example:**

class Engine

{ };

class Car

{

private:

Engine e; //Association

};

int main( void )

{

Car carobj;

return 0;

}

//Dependant Object : Car Object

//Dependancy Object : Engine Object

**\* Composition**

* If dependancy object do not exist without Dependant object then it represents composition.
* Composition represents tight coupling.
* Example: Human has-a heart.

**\* Composition Example**

class Heart

{ };

class Human

{

Heart hrt; //Association->Composition

};

int main( void )

{

Human h;

return 0;

}

//Dependant Object : Human Object

//Dependancy Object : Heart Object

**\* Aggregation**

* If dependancy object exist without Dependant object then it represents Aggregation.
* Aggregation represents loose coupling.

**\* Aggregation Example**

class Faculty

{ };

class Department

{

Faculty f; //Association->Aggregation

};

int main( void )

{

Department d;

return 0;

}

//Dependant Object : Department Object

//Dependancy Object : Faculty Object

**\* Inheritance**

* If "is-a" relationship exist between two types then we should use inheritance.
* Inheritance is also called as "Generalization".
* Example: Book is-a product
* During inheritance, members of base class inherit into derived class.
* If we create object of derived class then non static data members declared in base class get space inside it.
* Size of object = sum of size of non static data members declared in base class and derived class.
* If we use private/protected/public keyword to control visibility of members of class then it is called access specifier / mode of inheritance.
* If we use private/protected/public keyword to extend the class then it is called mode of inheritance.
* Default mode of inheritance is private.
  + Example: class Employee : person //is treated as class Employee : private Person
* Example: class Employee:public Person
* In all types of mode, private members inherit into derived class but we can not access it inside member function of derived class.
* If we want to access private members inside derived class then:
  + Either we should use member function(getter/setter).
  + or we should declare derived class as a friend inside base class.
* **Types of inheritance**
  1. Single Inheritance

**If single base class is having single derived class then it is called single inheritance.**

class A{};

class B : public A{};

* 1. Multiple Inheritance

**If multiple base classes are having single derived class then it is called multiple inheritance.**

**class** Base1

{

**public**:

**void** **fn**()

{

cout<<"Inside Base1";

}

};

**class** Base2

{

**public**:

**void** **fn1**()

{

cout<<"Inside Base2";

}

};

**class** multiple:**public** Base1, **public** Base2

{ };

* 1. Hierarchical Inheritance

**if single base class is having multiple derived classes then such inheritance is called hierarchical inheritance.**

class A{};

class B :public A{};

class C : public A{};

class D: public A{};

* 1. Multilevel Inheritance

**If single inheritance is having multiple levels then it is called multilevel inheritance**

**class** Base

{

**public**:

**void** **fn**()

{

cout<<"Inside Base Class";

}

};

**class** Derived:**public** Base

{ };

**class** Derived2 : **public** Derived

{ };

5.Hybrid Inheritance

**Combination of any two or more than two types of inheritance is called hybrid inheritance.**