Feb. 19 and 20

Concept on Inheritance

Classes are in relation to each other.

Derived class is a sub-type of base class.

Derived class extends its base class.

We can have objects of other class declared in the class.

Inheritance allows Derived Classes to inherit Base class data-members as well as member functions. Thus, if we call base class function with derived class instance, it would run perfectly.

They are synonym Super class – Sub class, Parent class – child class, and base class – derived class.

```
Eg: Class Person

{

Char arr[50];

string name; // string is also a type of class

Person ( string s): name(s)

{  strcpy (arr,s);   }

Public: void printName();

};

Class Student : public Person    // we use public around 95% times(mostly), while protected is used negligible and private is used around 5% only I some rare context.

{

float cgpa;

public:

        void printCGPA();

};
```

```
# include <iostream>
```

```
Using namespace std;
class Person
{ string name;
public:          Person() const {“ constructor person is called”;}
void printName() const { cout<<name<< endl;}
Person (string s) : name (s){}
~Person(){ cout<< “destructor of person “;}
};
Class Student: public Person {
          Float cgpa = 8.0;
          Public:
               Student(){ cout<<” Constructor for student is called”;}
               voidPrintCGPA() const{ cout<<cgpa<< endl;}
               ~Student(){ cout<< Destructor of student is called”<<endl;}


int main()
{
Student s;
S.name=” Mayank” // Will throw error( very basic, cannot modify private variable directly)
s.printCGPA();
}
```

OUTPUT:

```
Constructor of Person is called
Constructor of Student is called
8.0
Destructor of Student is called// Will print in reverse.
Destructor of Person is called
```

Usage of "final" keyword

Means you cannot derive a class anymore, meaning that **the "final" class** will always remain the leaf and never be the tree or a plant.

Protected data members:

Helps to modify the data members.

A protected variable of a parent or base class(When derived publicly) becomes the protected member of the derived class.

Data privacy violation: it happens when using public.

Most restricted is the private

There are three types of inheritance:

1. public 2. private 3. protected.

PS: Fundamental difference between friend class and inheritance:

Members are not inherited so a friend class can access member to its friend class but do not have the same data members from its class.

But in case of inheritance not only the data members can be accessed as well as they can be used.

Delegation: Its Without initialer list.

Invocation: Initializer list.

NOTE: All data members and member functions are directly inherited from parent class to derived class **except constructors and destructors.**

Class Relationship

It is broadly classified as: **Inheritance (is-a), Association (is related to) and Dependency (uses-a).**

Association is related to Aggregation and composition.

Inheritance types:

Public inheritance: Class D : public B{};

Protected inheritance: Class D : Protected B{};

Private inheritance: Class D : Private B{};

B: Base class

D: Derived Class

Note:

A private member in base class becomes inaccessible or hidden member in the derived class.

But a public member in the base class becomes public member in the derived class.

To use the same name for a function in the base and the derived class we need overloaded or overridden member functions.

Accessing private data members of base class in derived class:

if you want to access private data members of the base class, you can instantiate the constructor with an initializer list, so that when the object of base class is created default constructor is called of the base class, and also initializer list is executed at the same time when the object is created.

Hence if we want to use private members of base class in derived class we need to use initializer list in the default constructor or in the parameterized constructor.

NOTE: there are 5 member functions that are not inherited in the derived class: default constructor, parameter constructor, copy constructor, destructor and assignment operator.

They must be Redefined they are not inherited.

Delegation: derived member function delegates part of its duty to base class using class resolution operator (::) , while in **invocation** constructor of derived class calls the constructor of base class during initialization which does not require class resolution operator.

Protected Members:

It acts like a private member in base class & it is accessible in derived class and all classes derived from the derived class.

Protected member is inherited like a private member but is not hidden in derived class, in other words.

We see no definition of method inside the Derived Class. So, when we call derived class in the main function, the compiler first looks into the Derived Class for its definition. Then, it looks inside its Parent class, where it finds the definition and that version is called.

However, suppose we want to change the behavior of the inherited method inside our Derived Class. This feature provided by OOP is called **Overriding** but is technically correct to call redefinition.

Function Redefinition:

When a child class writes a function having the same signatures of the parent class, the child class overrides the parent class.

Function overloading: same name but different signatures.

Function overriding: same name and signature in the derived class overpowers precedence for subclass over superclass.

21-Feb & 23-Feb

Three Types of Inheritance:

Public inheritance is the most common type of derivation. C++ also allows other two types of derivation: Private and protected.

Hidden means not accessible to the public member functions of the derived class. It must be accessed by public member function of the base class.

Public Inheritance:

- Mostly used.
- Is a relation type Between base class and derived class.
- In other words, object of derived class is object of base class.
- Please note that: Private members of the base class remain private in the derived class. And they cannot be accessed through the public functions of the derived class. That is, they must be accessed through the member functions of the base class.

Private Inheritance:

- Less common than public inheritance Another piece of information represented.
- Public members of base class become private members in derived class.

- For example, stack is not a linked list, but it can inherit code from a linked list.
- Private inheritance does not define "a type" inheritance. It defines an implementation inheritance.

When is the initializer list important?

Invocation of constructor of parent.

Constant Initialization. Eg. One time initialization. It happens once.

List: Collection of items of same type.

Array: It is an example of a list. Basically, a concrete way of abstract data type. (ADT).

List is an ADT. That means one way of implementing list is by array.

Limitation of array? Size is the limitation. You cannot declare before hand.

Advantages of Linked List: It is dynamic, it can grow and subtract, addition and deletion is easier.

Major disadvantage: You cannot access to specific element, you need to traverse through all the other elements to reach out.

What about stack?

It can be implemented as an array and as a list.

Restriction? Just that ejection happens only from one side.

There are 2 types of Inheritance.

Interface and Implementation Inheritance.

Interface is when we use **Public** mode of Inheritance. (Use directly)

In this public remains public, protected remains protected and private remains inaccessible from base to derive class respectively.

Implementation Inheritance:

Protected:

public------protected

protected --------protected

private-------- Inaccessible. From base to derive class.

Private:

Public---------private

Protected--------- private

Private---------inaccessible

Whereas in Implementation Inheritance (Protected and Private mode of Inheritance) use it privately.

It is of “ is-a “ type.

Only C++ provides this sort of both Inheritance types, where as in Java and Python only public inheritance is used and is available.

Eg. Stack is not a linked list but it can inherit code from linked-list.

Linked List:

Background & Introduction

Problems with array: or you can say advantages of linked list over array

1. We need to pre-allocate the space. Space allocated is fixed. This can be solved by dynamic sized array. Eg. Vectors in C++, array list in JAVA and list in Python. They will double the size in typical implementation whenever they reach their maximum limit. And they copy the previous elements is transferred in the new double allocated sized…But the problem in vectors will be Big O(n).
2. The size of the arrays is fixed, so we must know the upper limit on the number of elements in advance. Linked lists are dynamic and the size of the linked list can be incremented or decreased during runtime.
3. Inserting a new element in an array of elements is expensive, because a room must be created for them, and to create room, existing elements must shift. For example, in a system, if we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040].

And if we want to insert a new ID 1005, then to maintain the sorted order, we must move all the elements after 1000 (excluding 1000). Deletion is also expensive with arrays unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved. On the other hand, nodes in linked lists can be inserted or deleted without any shift operation and is efficient than that of arrays.

Linked list: they are linear or sequential data structures in which elements are stored at non-contagious memory locations and are linked to each other using pointers.

Each element in linked list consists of 2 parts: **DATA and NEXT pointer.**

Data: It stores the data value of the node i.e information to be stored at the current node.

Next pointer: Pointer variable which stores the address of the next node in the memory.

Disadvantages of Linked List:

1. Random access is not allowed in Linked Lists. We have to access elements sequentially starting from the first node. So, we cannot do a binary search with linked lists efficiently with its default implementation. Therefore, lookup or search operation is costly in linked lists in comparison to arrays.
2. Extra memory space for a pointer is required with each element of the list.
3. Not cache-friendly. Since array elements are present at contiguous locations, there is a locality of reference not there for linked lists.

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head node of the list. If the linked list is empty, then the value of the head node is NULL.

In C/C++, we can represent a node using structure. Below is an example of a linked list node with integer data.

```
struct Node
{
   int data;
   struct Node* next;
};
```

Or using class as well:

class Node { public: int data; Node* next; // Constructor Node(int d) : data(d), next(nullptr) {} }

26-27February

A base class type reference or Pointer can refer to derived class object.

Here is the example:

```
Class Base{};

Class Derived: public Base{};

int main()

{

Base *b = new Derived;
```

Derived d;

Base &b = d; // above 3 statements are valid in C++

Return 0;

}

Polymorphism gives us the ability to write several versions of a function in different classes. We can define a pointer (or a reference) that can point to the base class; we can then let the pointer point to any object in the hierarchy. For this reason, the pointer and reference variables are sometimes called polymorphic.

For polymorphism, we need pointers (or references), we need exchangeable objects, and we need virtual functions.

Run Time Polymorphism

In run-time polymorphism, the decision of which function to call is determined at runtime based on the actual object type rather than the reference or pointer type. It is also known as Dynamic Polymorphism because the function calls are dynamically bonded at the runtime.

Run Time Polymorphism can be exhibited by:

Method Overriding using Virtual Functions

Method Overriding

Method overriding refers to the process of creating a new definition of a function in a derived class that is already defined inside its base class. Some rules that must be followed while overriding a method are:

Method names must be the same.

Method parameters must be the same.

Virtual Function

Virtual Function is a member function that is declared as virtual in the base class and it can be overridden in the derived classes that inherit the base class.

Virtual functions are generally declared in the base class and are typically defined in both the base and derived classes.

Difference Between Compile Time And Run Time Polymorphism

Compile-Time Polymorphism	**Run-Time Polymorphism**
It is also called Static Polymorphism.	It is also known as Dynamic Polymorphism.
In compile-time polymorphism, the compiler determines which function or operation to call based on the number, types, and order of arguments.	In run-time polymorphism, the decision of which function to call is determined at runtime based on the actual object type rather than the reference or pointer type.
Function calls are statically binded.	Function calls are dynamically binded.
Compile-time Polymorphism can be exhibited by: 1. Function Overloading 2. Operator Overloading	Run-time Polymorphism can be exhibited by Function Overriding.
Faster execution rate.	Comparatively slower execution rate.
Inheritance in not involved.	Involves inheritance.

NOTE: When a function is defined as virtual, all functions in the hierarchy of classes with the same signature are automatically virtual. There is no necessity to put virtual in all functions.

Working of Virtual Functions: How do they work internally.

C++ standard does not say anything about implementation of virtual functions, how should they be implemented by compilers.

Compilers use 2 things to implement virtual function inside them. One is VPTR which is maintained withn every object and other is vtable, which is there for every class.

And this vptr points to vtable .

When you create an object of a class, the compiler add some code to the constructor.

This code sets VPTR to vtable of the class. Vtable of the class stores address of all virtual functions Present in the class.

So when you make call to an overridden function, Your compiler goes to vptr of that object , and using the VPTR, it finds the vtable of the class, and using the vtable it finds the address of the function to be called.

So, these virtual functions they do extra thing, maintaining VPTR of every object.

They maintain Vtable of every class. and this Vtable stores address of all virtual functions.

They add some extra code at 2 places, one code in the constructor of the object that sets vptr to vtable. One code when you make overiden function calls. All this extra work is done by compilers, and this causes some extra CPU cycles.

Virtual functions have extra advantages since they reduce code complexity.

We can use them unless we have time restricted environment.

VPTR AND VTABLE:

In polymorphism, the system creates a virtual table for each class, each entry in each vtable has a pointer to the corresponding virtual function.

When ptr is pointing to the Base object, the VPTR pointer added to the Base class object is reached, which is pointing to the vtable of the Base class. In this case, the vtable has only one entry, which invokes the only virtual function in Base class. When ptr is pointing to the Derived object, the vtable of the Derived object is reached, and the print function defined in the Derived class is invoked.

You will see how instances of derived classes are often stored using base class pointers and why this is significant.

regardless of how an instance is stored (as its own type or as that of a base class), the correct version of a virtual function will always be applied through dynamic binding.

A derived class object pointed by a base class pointer through upcasting.

derived class instances may be pointed to by base class objects (that is, by pointers of a base class type).

In this, Functions in the derived class with the same name and signature of a virtual function in any ancestor class redefine the virtual function in those base classes. Here, the keyword virtual is optional in the derived class prototype.

• Optionally and preferred, the keyword override can be added as part of the extended signature in the derived class prototype. This recommended practice will allow the compiler to flag an error if the signature of the intended overridden method does not match the signature as specified in the base class. The override keyword can eliminate unintended function hiding.

Functions with the same name, yet a different signature in a derived class, do not redefine a virtual function in their base class; rather, they hide the methods found in their base classes.

You will see how instances of derived classes are often stored using base class pointers and why this is significant.

regardless of how an instance is stored (as its own type or as that of a base class), the correct version of a virtual function will always be applied through dynamic binding.

A derived class object pointed by a base class pointer through upcasting.

derived class instances may be pointed to by base class objects (that is, by pointers of a base class type).

In this, Functions in the derived class with the same name and signature of a virtual function in any ancestor class redefine the virtual function in those base classes. Here, the keyword virtual is optional in the derived class prototype.

• Optionally and preferred, the keyword override can be added as part of the extended signature in the derived class prototype. This recommended practice will allow the compiler to flag an error if the signature of the intended overridden method does not match the signature as specified in the base class. The override keyword can eliminate unintended function hiding.

Functions with the same name, yet a different signature in a derived class, do not redefine a virtual function in their base class; rather, they hide the methods found in their base classes.

11-March and 12-March

Standard library Templates, Exception handling, Vectors:

As we have told you in our first lab lecture that C++ allows use of generic programming concept which is using C++ templates. (Template functions and Template classes).

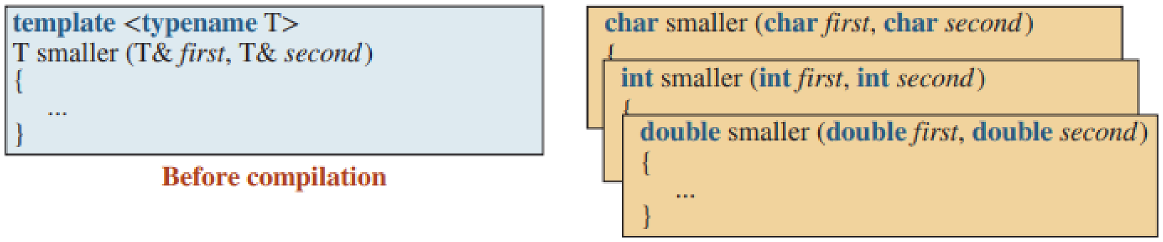
Template is basically used for reusability.

Using function template requires 2 things: Syntax and Instantiation.

The template header contains the keyword template followed by a list of symbolic types inside two angle brackets <>. The template function header follows the rules for function headers except that the types are symbolic as declared in the template header. While multiple generic types are possible, a function template with more than two generic types is rare. Some older code uses the term class rather than typename.

Instantiation Using template functions postpones the creation of non template function definitions until compilation time. This means that when a program involving a function template is compiled, the compiler creates as many versions of the function as needed by function calls. This process is referred to as template instantiation, but it should not be confused with instantiation of an object from a type.

Ex: Check the example of swap function.



```
template <typename T>
T smaller (T& first, T& second)
{
    ...
}
```

Before compilation

```
char smaller (char first, char second)
{
int smaller (int first, int second)
{
double smaller (double first, double second)
{
    ...
}
```

After compilation

A function template is instantiated into several functions.

Try working with a program that uses template function to print elements of an array of any type.

NOTE: If we try to call the function template to find the smaller between an integer and a floating-point value, we get an error message. In that case we can use explicit type conversion.

Ex: cout << smaller (23, 67.2); // Errors! Two different types for the same template type T.

cout << smaller < double > (23, 67.2); // 23 will be changed to 23.0

STL: It provides library for implementation of basic data structures and basic algorithms. It is never recommended and not a good idea that you implement your own algorithms when it is asked to solve a bigger or complex problems. For example: sorting or doing binary search as a basic step in a big problem.

STL can be divided into 2 parts: Containers and algorithms.

Exception handling:

It is an unusual condition that might happen when your program is running.

More examples of unusual conditions are:

Divide by zero.

No heap memory available. The heap area is full.

Accessing array elements outside. Array out of bound.

Pop from an empty stack.