Data Hiding, Encapsulation, Polymorphism | L:23 | C++ |

==========

Data Hiding

---

1. Data hiding is an object-oriented programming technique for protecting the data within a class from unwanted access and preventing unneeded intrusion from outside the class.
2. It helps hide internal object details, i.e., data members within the class, to prevent its direct access from outside the class.
3. Data hiding can be achieved through the encapsulation, as encapsulation is a subprocess of data hiding.
4. Private, public and protected ⇒ 3 access specifiers are available in the class.
5. Usually the data within the class are private and functions are public.

6. Private members/methods: Functions and variables declared as private can be accessed only within the same class, and they cannot be accessed outside the class they are declared.
7. Public members/methods: Functions and variables declared under public can be accessed from anywhere.
8. Protected members/methods: Functions and variables declared as protected cannot be accessed outside the class except a child class. This specifier is generally used in inheritance.

Example

==========

```
#include <iostream>
using namespace std;
class A{
    private:
        int val;
    public:
        void setVal(int val)
        {
            this->val=val;
        }
        int getVal()
        {
            return val;
        }
};
```

```
int main()
{
    A obja;
    obja.setVal(10);
    cout<<obja.getVal()<<endl;//10
    return 0;
}
```

In this example, there is a class with a variable and two functions. Here, the variable "val" is private.
So, it can be accessed only by the members of the same class, and it can't be accessed anywhere else.
Hence, it is unable to access this variable outside the class, which is called data hiding.
==========

Encapsulation
==========

1. Encapsulation is a process of combining member functions and data members in a single unit called a class. ⇒ **class = methods+variables**
2. **The purpose is to prevent access to the data directly.**
3. Access to them is provided through the functions of the class.
4. It is one of the popular features of Object-Oriented Programming(OOPs), which helps in **data hiding.**
5. **Encapsulation also leads to data abstraction.**

Two Important property of Encapsulation

1. **Data Protection:** Encapsulation protects the internal state of an object by keeping its data members private. Access to and modification of these data members is restricted to the class's public methods, ensuring controlled and secure data manipulation.
2. **Information Hiding:** Encapsulation hides the internal implementation details of a class from external code.

Features of Encapsulation
---

1. We can not access any function from the class directly. We need an object to access that function that is using the member variables of that class.
2. The function which we are making inside the class must use only member variables, only then it is called encapsulation.
3. If we don't make a function inside the class which is using the member variable of the class then we don't call it encapsulation.
4. Encapsulation improves readability, maintainability, and security by grouping data and methods together.
5. It helps to control the modification of our data members.

Example 1- Encapsulation

```cpp
// C++ program to demonstrate Encapsulation
#include <iostream>
using namespace std;
class Encapsulation
{
private:
    // Data hidden from outside world since x is private
    int x;
public:
    // Function to set value of variable x
    void set(int a)
    {
        x = a;
    }
    // Function to return value of variable x
    int get()
    {
        return x;
    }
};
// Driver code
int main()
{
    Encapsulation obj;
    obj.set(5);
    cout << obj.get();
    return 0;
}
```

Explanation:

In the above program, the variable x is made private. This variable can be accessed and manipulated only using the functions get() and set() which are present inside the class. Thus we can say that here, the variable x and the functions get() and set() are bound together which is nothing but encapsulation.

Role of Access Specifiers in Encapsulation

1. Access specifiers facilitate Data Hiding in C++ programs by restricting access to the class member functions and data members.

2. There are three types of access specifiers in C++:
    a. **Private:** Private access specifier means that the member function or data member can only be accessed by other member functions of the same class.
    b. **Protected:** A protected access specifier means that the member function or data member can be accessed by other member functions of the **same class or by derived classes.**
    c. **Public:** Public access specifier means that the member function or data member can be accessed by any code.

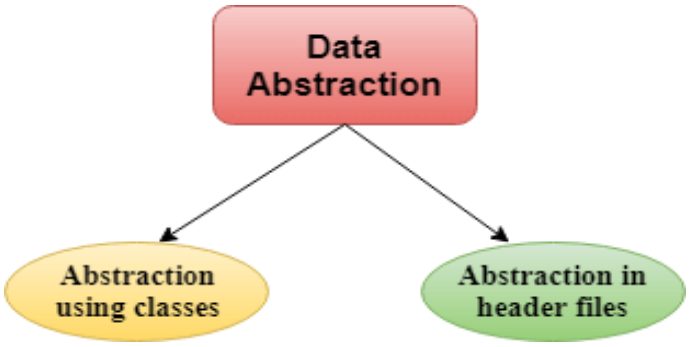
3. By default, all data members and member functions of a class are made private by the compiler.

==========

Abstraction ⇒ Hide the Implementations and provide only important details

---

1. Data abstraction is one of the most essential and important features of object-oriented programming in C++.
2. Abstraction means displaying only essential information and hiding the implementation details.
3. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.
4. Data Abstraction can be achieved in two ways:
    a. **Abstraction using classes**
    b. **Abstraction in header files.**



---

5. **Abstraction using classes:** An abstraction can be achieved using classes. A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.

6. Abstraction in header files: Another type of abstraction is header file. For example, the **pow() function** available is used to calculate the power of a number without actually knowing which algorithm function is used to calculate the power. Thus, we can say that header files hide all the implementation details from the user.

7. **Access Specifiers Implement Abstraction:**
    a. **Public specifier:** When the members are declared as public, members can be accessed anywhere from the program.
    b. **Private specifier:** When the members are declared as private, members can only be accessed only by the member functions of the class.

We can easily implement abstraction using the above two features provided by access specifiers.
The members that define the internal implementation can be marked as private in a class.
The important information needed to be given to the outside world can be marked as public and these public members can access the private members as they are inside the class.

Example
========

```
// C++ Program to Demonstrate the working of Abstraction
#include <iostream>
using namespace std;

class Abstraction {
private:
    int a, b;

public:
    // method to set values of private members
    void set(int x, int y)
    {
        a = x;
        b = y;
    }

    void display()
    {
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
    }
};
```

```
int main()
{
    Abstraction obj;
    obj.set(10, 20);
    obj.display();
    return 0;
}
```

—----------

We can see in the above program we are not allowed to access the **variables a** and **b** directly. However, we can call the function **set()** to set the values in **a and b** and the function **display**() to display the values of **a and b.**

==========

Difference between Abstraction and Encapsulation:

Abstraction:

1. Abstraction involves hiding unnecessary details and exposing only relevant information or functionalities.
2. Implemented using abstract classes and interfaces in languages like C++.

Encapsulation:

1. Encapsulation involves bundling data (attributes) and methods (functions) that operate on that data into a single unit (class) and controlling access to them.
2. It hides the internal state of an object and provides controlled access using access modifiers (private, protected, public).

Example of Abstraction:

—----------

```
#include <iostream>
using namespace std;
class Summation {
private:
    // private variables
    int a, b, c;
public:
    void sum(int x, int y)
    {
        a = x;
        b = y;
        c = a + b;
        cout<<"Sum of the given number is : "<<c<<endl;
```

```
    }
};
int main()
{
    Summation s;
    s.sum(5, 4);
    return 0;
}
```

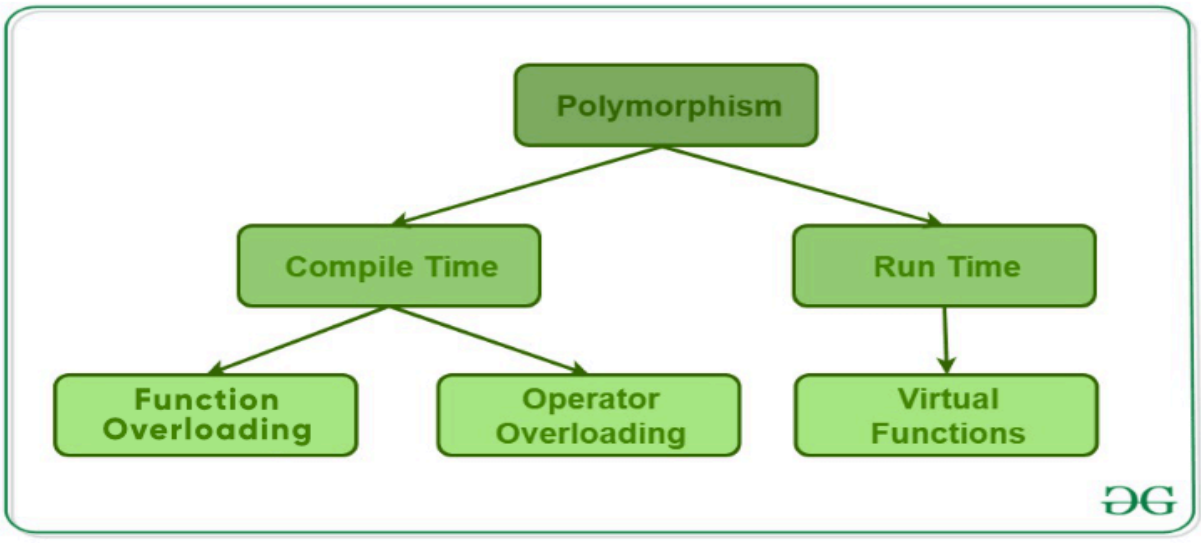
In this example, we can see that abstraction is achieved by using class. The class 'Summation' holds the private members a, b and c, which are only accessible by the member functions of that class.

==========

Polymorphism

==========

1. The word polymorphism means having many forms.
2. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
3. Real life examples of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.
4. Polymorphism is considered as one of the important features of Object Oriented Programming.
5. In C++ polymorphism is mainly divided into two types:
    a. **Compile time Polymorphism**
    b. **Runtime Polymorphism**



Compile-Time Polymorphism

This type of polymorphism is achieved by **function overloading** or **operator overloading.**

A. Function Overloading

1. When there are multiple functions with the same name but different parameters, then the functions are said to be overloaded, hence this is known as Function Overloading.
2. Functions can be overloaded by changing the number of arguments or/and changing the type of arguments.
3. In simple terms, it is a feature of object-oriented programming providing many functions that have the same name but distinct parameters when numerous tasks are listed under one function name. There are certain Rules of Function Overloading that should be followed while overloading a function.

Example: Function Overloading

==========

```
// C++ program to demonstrate function overloading or Compile-time
Polymorphism
#include <bits/stdc++.h>
using namespace std;
class Test {
public:
    // Function with 1 int parameter
    void func(int x)
    {
        cout<<"value of x is: "<< x << endl;
    }
    // Function with same name but 1 double parameter
    void func(double x)
    {
        cout<<"value of x is: "<< x << endl;
    }
    // Function with same name and 2 int parameters
    void func(int x, int y)
    {
        cout<<"value of x and y is " << x << ", " << y<<endl;
    }
};
int main()
{
    Test obj1;
```

```
    // Function being called depends on the parameters passed
    // func() is called with int value
    obj1.func(7);//value of x is: 7

    // func() is called with double value
    obj1.func(9.132);//value of x is: 9.132

    // func() is called with 2 int values
    obj1.func(85, 64);//value of x and y is 85, 64
    return 0;
}
```

—----------

Explanation: In the above example, a single function named function func() acts differently in three different situations, which is a property of polymorphism.

—----------

B. Operator Overloading

—----------

1. C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.
2. Operator overloading is a compile-time polymorphism.
3. For example, we can make the operator (‘+’) for string class to concatenate two strings.
4. We know that this is the addition operator whose task is to add two operands.
5. So a single operator ‘+’ when placed between integer operands , adds them and when placed between string operands, concatenates them.

Example:

—----------

```
//  C++  program  to  demonstrate  operator  overloading  or  Compile-time
Polymorphism
#include <bits/stdc++.h>
using namespace std;
class ComplexNumber
{
    private:
        double real;
        double imag;
    public:
        ComplexNumber(double r=0,double i=0)
        {
            this->real=r;
            this->imag=i;
```

```
        }
        //overloading the + operator
        ComplexNumber operator+(const ComplexNumber &obj) const
        {
            ComplexNumber result;
            result.real=real+obj.real;
            result.imag=imag+obj.imag;
            return result;
        }
        void print()
        {
            cout<<real<<"+i"<<imag<<endl;
        }
};
int main()
{
    ComplexNumber num1(1,2);
    ComplexNumber num2(2,3);
    ComplexNumber sum=num1+num2;
    sum.print();
    return 0;
}
```

—----------

Explanation: In the above example, the operator '+' is overloaded. Usually, this operator is used to add two numbers (integers or floating point numbers), but here the operator is made to perform the addition of two imaginary or complex numbers.

—----------

Difference between Operator Functions and Normal Functions

---

1. Operator functions are the same as normal functions.
2. The only differences are, that the name of an operator function is always the operator keyword followed by the symbol of the operator.
3. Operator functions are called when the corresponding operator is used.

==========

Run-Time Polymorphism

---

1. Runtime polymorphism, also known as late binding or dynamic polymorphism, is achieved through function overriding.
2. In runtime polymorphism, the function call resolution occurs at runtime, meaning the appropriate function to be executed is determined based on the actual object type at runtime.

3. On the other hand, compile-time polymorphism involves the compiler determining which function calls to bind to the object during compilation itself, not at runtime.

A. Function Overriding

---

1. Function Overriding occurs when a derived class has a definition for one of the member functions of the base class.
2. That base function is said to be overridden.

Example:

—----------

```
#include <bits/stdc++.h>
using namespace std;
class B
{
    public:
        void fun()
        {
            cout<<"fun in the base class"<<endl;
        }
        // virtual void fun() //to do the late binding of the fun
        // {
        //     cout<<"fun in the base class"<<endl;
        // }
};
class D:public B
{
    public:
        void fun()
        {
            cout<<"fun in the derived class"<<endl;
        }
};
int main()
{
    B objb;
    D objd;
    objb.fun();//fun in the base class
    objd.fun();//fun in the derived class

    D *dptr=&objd;
    dptr->fun();//fun in the derived class
```

```
    B *bptr1=&objb;
    bptr1->fun();//fun in the base class

    B *bptr2=&objd;
    bptr2->fun();//fun in the base class
    //Here we are storing the address of obj of the derived class into the
pointer of Base class
    //but fun() in the base class will be called.
    //to solve this problem we use the virtual function
    //virtual function ==> it is used to do the late binding
     //If we declare the fun() in the base class as virtual then fun() in
the derived class will be executed.
    return 0;
}
```

—----------

B. Virtual Function

—----------

To read about the virtual function ⇒

==========

Virtual Class and methods(L-26,27)

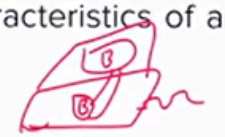
https://docs.google.com/document/d/1o3tImuhZKoz9uypAQ3z65L04DwHmdCizWqxzHYONBrw/edit?usp=sharing

==========End of Polymorphism==========

.....

Abstraction

- Abstraction is primarily used to hide the complexity.
- It indicates the necessary characteristics of an object that differentiates it from all other types of objects.



- An abstraction concentrates on the external aspect of an object. For an object, abstraction provides the separation of the crucial behaviour from its implementation.
- A proper abstraction emphasizes on the details that are important for the reader or user and suppresses features that are irrelevant and deviant.

21:30 / 54:10

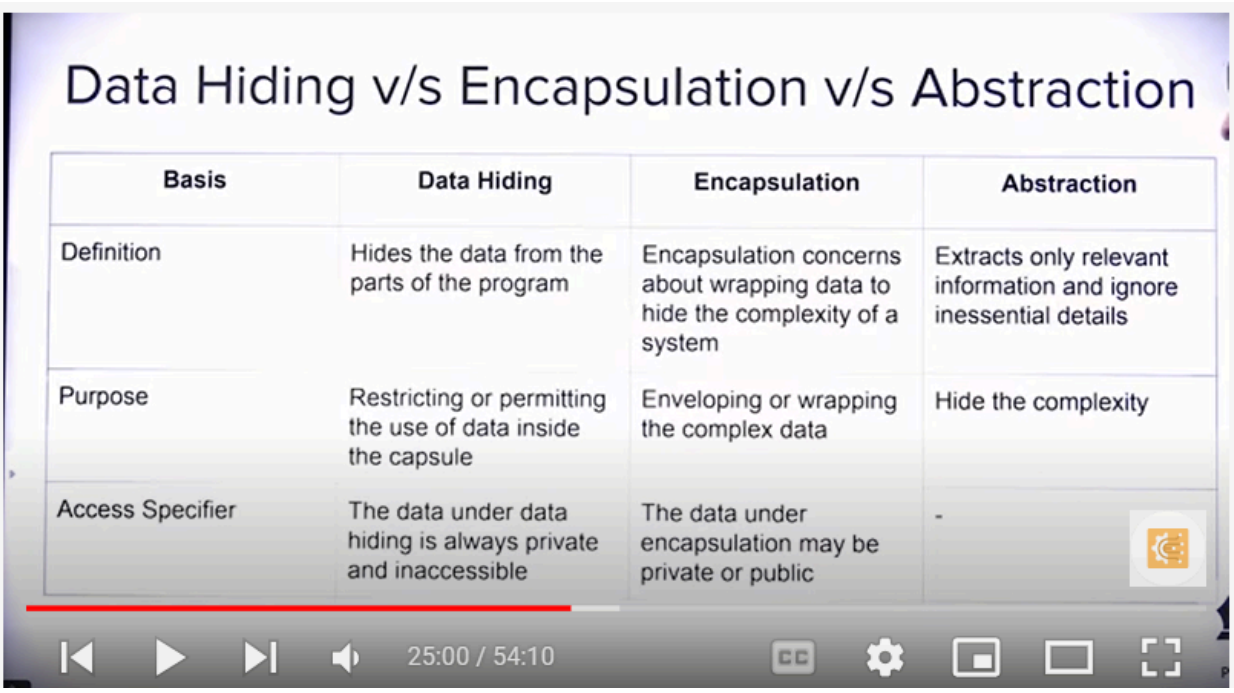
.....

Types of Abstraction

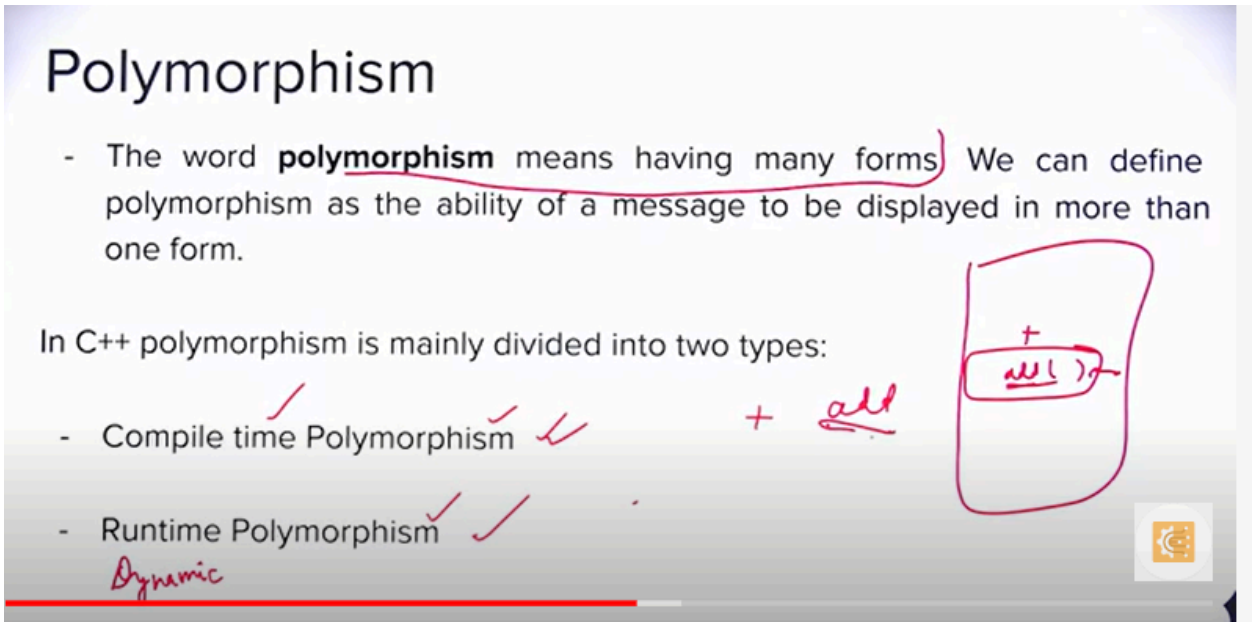
- Procedural abstraction – It includes series of the instructions having the specified functions. Procedural abstraction provides mechanisms for abstracting well defined procedures or operations as entities. The implementation of the procedure requires a number of steps to be performed.
- Data abstraction – It is set of data that specifies and describes a data object. This principle is at the core of Object Orientation. In this form of abstraction, instead of just focusing on operations, we focus on data first and then the operations that manipulate the data.

24:25 / 54:10

.....


Data Hiding v/s Encapsulation v/s Abstraction
Basis
Data Hiding
Encapsulation
Abstraction
Definition
Hides the data from the parts of the program
Encapsulation concerns about wrapping data to hide the complexity of a system
Extracts only relevant information and ignore inessential details
Purpose
Restricting or permitting the use of data inside the capsule
Enveloping or wrapping the complex data
Hide the complexity
Access Specifier
The data under data hiding is always private and inaccessible
The data under encapsulation may be private or public
-
25:00 / 54:10

.....


Polymorphism
- The word polymorphism means having many forms. We can define polymorphism as the ability of a message to be displayed in more than one form.
In C++ polymorphism is mainly divided into two types:
- Compile time Polymorphism
- Runtime Polymorphism
Dynamic
+ add
+
add( )

.....