Unit 4 - Pointers:

Content :- Pointers: Pointer to Derived Class, array of Pointers, Inheritance and Polymorphism: Inheritance, Class hierarchy, derivation, public, private & protected, abstract Classes, Single, Multilevel, Multiple, Hierarchical, Hybrid, benefits of Inheritance.

Overall, these additional outcomes further enhance students' understanding and proficiency in using inheritance and polymorphism effectively in software development projects, preparing them for real-world applications in the field of computer science and programming.

C++ Pointers

The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.
The symbol of an address is represented by a pointer. In addition to creating and modifying dynamic data structures, they allow programs to emulate call-by-reference. One of the principal applications of pointers is iterating through the components of arrays or other data structures. The pointer variable that refers to the same data type as the variable you're dealing with has the address of that variable set to it (such as an int or string).

Syntax

1. datatype *var_name;
2. **int** *ptr; // ptr can point to an address which holds int data

How to use a pointer?

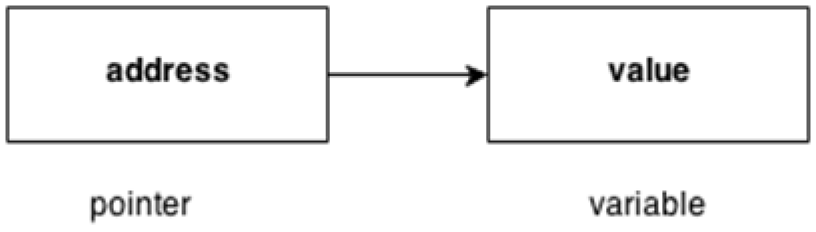
1. Establish a pointer variable.
2. employing the unary operator (&), which yields the address of the variable, to assign a pointer to a variable's address.
3. Using the unary operator (*), which gives the variable's value at the address provided by its argument, one can access the value stored in an address.

Since the data type knows how many bytes the information is held in, we associate it with a reference. The size of the data type to which a pointer points is added when we increment a pointer.

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

Unit 4 - Pointers:

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
∗ (asterisk sign)	Indirection operator	Access the value of an address.



Advantage of pointer

1) Pointer reduces the code and improves the performance, it is used to retrieving strings, trees etc. and used with arrays, structures and functions.
2) We can return multiple values from function using pointer.
3) It makes you able to access any memory location in the computer's memory.

Usage of pointer

There are many usage of pointers in C++ language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.

2) Arrays, Functions and Structures

Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

Symbols used in pointer

Declaring a pointer

The pointer in C++ language can be declared using ∗ (asterisk symbol).

```
int ∗   a; //pointer to int
char ∗  c; //pointer to char
```

Pointer Example

Let's see the simple example of using pointers printing the address and value.

```
#include <iostream>
using namespace std;
```

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

Unit 4 - Pointers:

```
int main()
{
int number=30;
int ∗   p;
p=&number;//stores the address of number variable
cout<<"Address of number variable is:"<<&number<<endl;
cout<<"Address of p variable is:"<<p<<endl;
cout<<"Value of p variable is:"<<*p<<endl;
   return 0;
}
```

Output:

```
Address of number variable is:0x7ffccc8724c4
Address of p variable is:0x7ffccc8724c4
Value of p variable is:30
```

Pointer Program to swap 2 numbers without using 3rd variable

```
#include <iostream>
using namespace std;
int main()
{
int a=20,b=10,∗p1=&a,∗p2=&b;
cout<<"Before swap: ∗p1="<<∗p1<<" ∗p2="<<∗p2<<endl;
∗p1=∗p1+∗p2;
∗p2=∗p1-∗p2;
∗p1=∗p1-∗p2;
cout<<"After swap: ∗p1="<<∗p1<<" ∗p2="<<∗p2<<endl;
   return 0;
}
```

Output:

```
Before swap: ∗p1=20 ∗p2=10
After swap: ∗p1=10 ∗p2=20
```

C++ Inheritance

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

Unit 4 - Pointers:

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

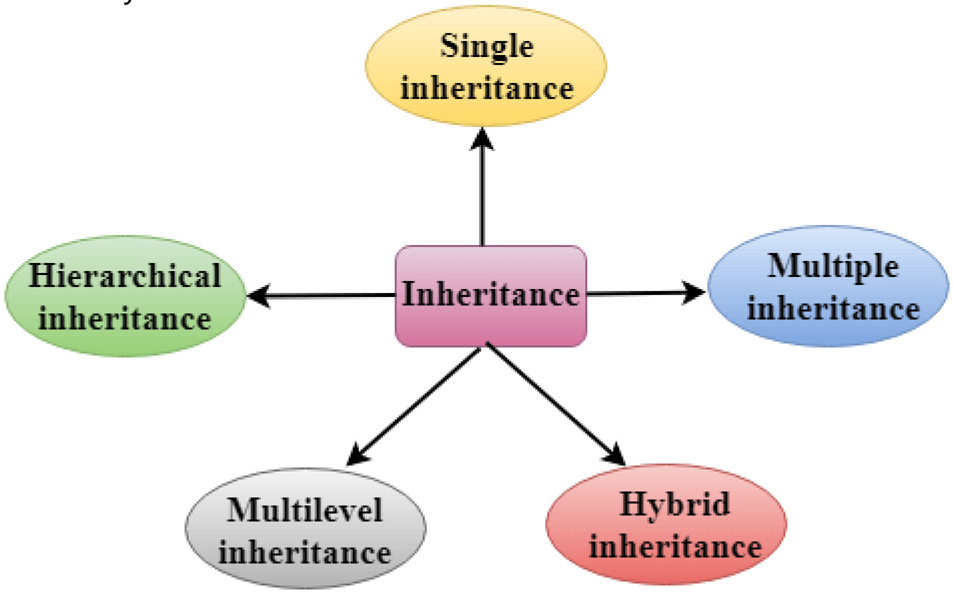
Advantage of C++ Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

Types Of Inheritance

C++ supports five types of inheritance:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance



Derived Classes

A Derived class is defined as the class derived from the base class.
The Syntax of Derived class:

```
1. class derived_class_name :: visibility-mode base_class_name
2. {
3.     // body of the derived class.
4. }
```

Where,
derived_class_name: It is the name of the derived class.

Unit 4 - Pointers:

visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.
base_class_name: It is the name of the base class.

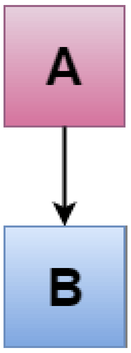
- When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.
- When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

Note:

- In C++, the default mode of visibility is private.
- The private members of the base class are never inherited.

C++ Single Inheritance

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



Where 'A' is the base class, and 'B' is the derived class.

C++ Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```
#include <iostream>
using namespace std;
 class Account {
   public:
   float salary = 60000;
 };
   class Programmer: public Account {
   public:
```

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

```
    float bonus = 5000;
  };
int main(void) {
    Programmer p1;
    cout<<"Salary: "<<p1.salary<<endl;
    cout<<"Bonus: "<<p1.bonus<<endl;
   return 0;
}
```

Output:

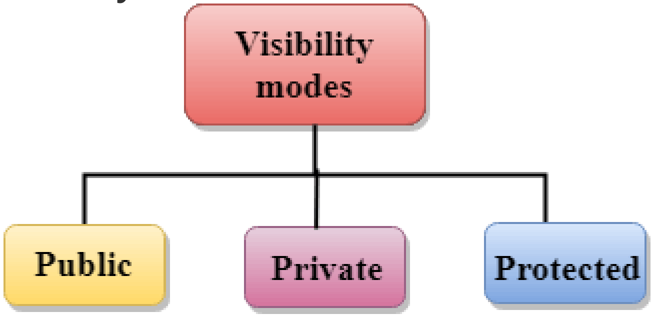
```
Salary: 60000
Bonus: 5000
```

How to make a Private Member Inheritable

The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.

C++ introduces a third visibility modifier, i.e., **protected**. The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.

Visibility modes can be classified into three categories:



- **Public**: When the member is declared as public, it is accessible to all the functions of the program.
- **Private**: When the member is declared as private, it is accessible within the class only.
- **Protected**: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

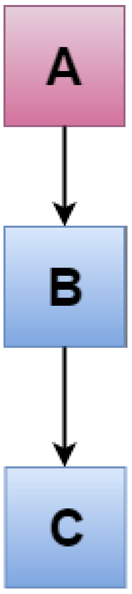
Unit 4 - Pointers:

Visibility of Inherited Members

Base class visibility	**Derived class visibility**		
	Public	**Private**	**Protected**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

C++ Multilevel Inheritance

Multilevel inheritance is a process of deriving a class from another derived class.



Unit 4 - Pointers:

C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C++.

```
#include <iostream>
using namespace std;
 class Animal {
   public:
 void eat() {
     cout<<"Eating..."<<endl;
 }
   };
   class Dog: public Animal
   {
       public:
     void bark(){
    cout<<"Barking..."<<endl;
     }
   };
   class BabyDog: public Dog
   {
       public:
     void weep() {
    cout<<"Weeping...";
     }
   };
int main(void) {
    BabyDog d1;
    d1.eat();
    d1.bark();
     d1.weep();
     return 0;
}
```

Output:

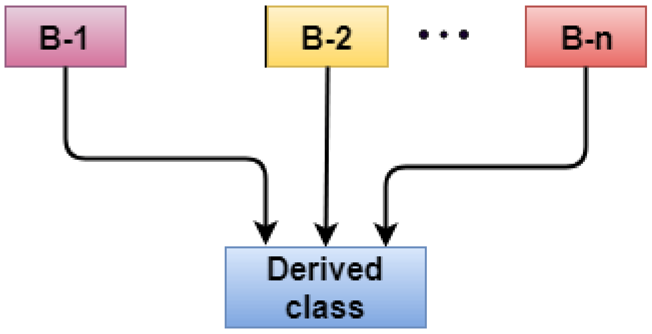
```
Eating...
```

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

```
Barking...
Weeping...
```

C++ Multiple Inheritance

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



Syntax of the Derived class:

```
class D : visibility B-1, visibility B-2, ?
{
    // Body of the class;
}
```

Let's see a simple example of multiple inheritance.

```
#include <iostream>
using namespace std;
class A
{
    protected:
     int a;
    public:
    void get_a(int n)
    {
        a = n;
    }
};

class B
{
    protected:
    int b;
    public:
```

Unit 4 - Pointers:

```
    void get_b(int n)
    {
        b = n;
    }
};
class C : public A,public B
{
   public:
    void display()
    {
        std::cout << "The value of a is : " <<a<< std::endl;
        std::cout << "The value of b is : " <<b<< std::endl;
        cout<<"Addition of a and b is : "<<a+b;
    }
};
int main()
{
   C c;
   c.get_a(10);
   c.get_b(20);
   c.display();

    return 0;
}
```

Output:

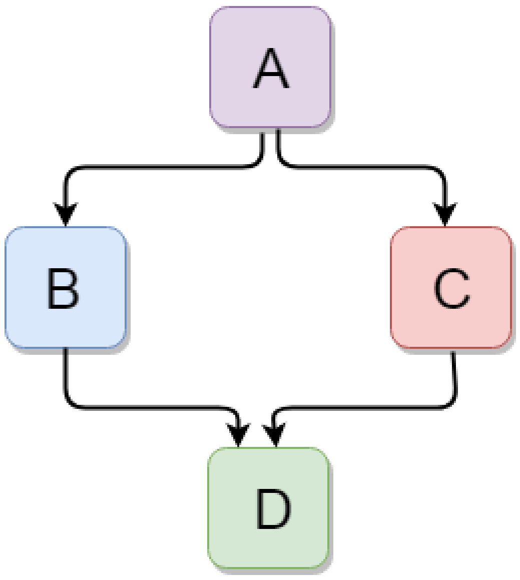
```
The value of a is : 10
The value of b is : 20
Addition of a and b is : 30
```

In the above example, class 'C' inherits two base classes 'A' and 'B' in a public mode.

C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.

Unit 4 - Pointers:



Let's see a simple example:

```
#include <iostream>
using namespace std;
class A
{
   protected:
   int a;
   public:
   void get_a()
   {
     std::cout << "Enter the value of 'a' : " << std::endl;
     cin>>a;
   }
};

class B : public A
{
   protected:
   int b;
   public:
   void get_b()
   {
      std::cout << "Enter the value of 'b' : " << std::endl;
     cin>>b;
```

Unit 4 - Pointers:

```
    }
};
class C
{
    protected:
    int c;
    public:
    void get_c()
    {
        std::cout << "Enter the value of c is : " << std::endl;
        cin>>c;
    }
};

class D : public B, public C
{
    protected:
    int d;
    public:
    void mul()
    {
         get_a();
         get_b();
         get_c();
         std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
    }
};
int main()
{
    D d;
    d.mul();
    return 0;
}
```

Output:

ADVERTISEMENT

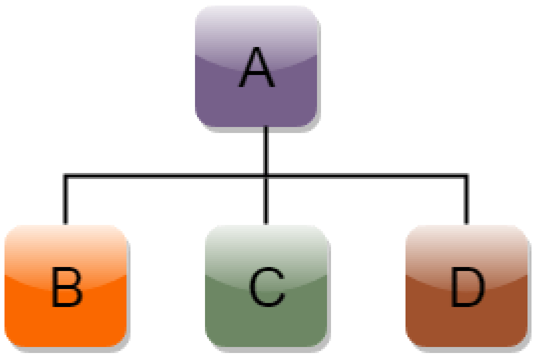
```
Enter the value of 'a' :
```

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

```
10
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000
```

C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



Syntax of Hierarchical inheritance:

```
class A
{
    // body of the class A.
}
class B : public A
{
    // body of class B.
}
class C : public A
{
    // body of class C.
}
class D : public A
{
    // body of class D.
}
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Shape                 // Declaration of base class.
```

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

```
{
    public:
    int a;
    int b;
    void get_data(int n,int m)
    {
        a= n;
        b = m;
    }
};
class Rectangle : public Shape  // inheriting Shape class
{
    public:
    int rect_area()
    {
        int result = a*b;
        return result;
    }
};
class Triangle : public Shape    // inheriting Shape class
{
    public:
    int triangle_area()
    {
        float result = 0.5*a*b;
        return result;
    }
};
int main()
{
    Rectangle r;
    Triangle t;
    int length,breadth,base,height;
    std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
    cin>>length>>breadth;
    r.get_data(length,breadth);
    int m = r.rect_area();
    std::cout << "Area of the rectangle is : " <<m<< std::endl;
    std::cout << "Enter the base and height of the triangle: " << std::endl;
    cin>>base>>height;
    t.get_data(base,height);
    float n = t.triangle_area();
    std::cout <<"Area of the triangle is : "  << n<<std::endl;
    return 0;
}
```

Output:

```
Enter the length and breadth of a rectangle:
23
20
Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5
```

C++ Polymorphism

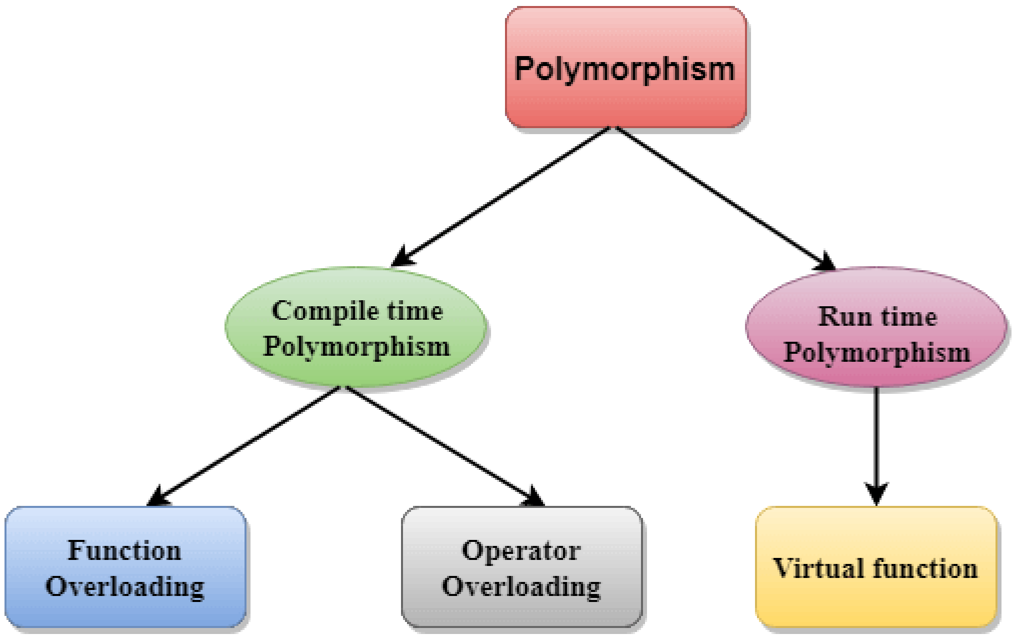
The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.

Real Life Example Of Polymorphism

Let's consider a real-life example of polymorphism. A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

There are two types of polymorphism in C++:

Unit 4 - Pointers:



-
- **Compile time polymorphism**: The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

```
  class A                               //  base class declaration.
 {
    int a;
    public:
    void display()
    {
        cout<< "Class A ";
     }
 };
class B : public A                      //  derived class declaration.
{
  int b;
  public:
 void display()
 {
    cout<<"Class B";
```

17.   }

18. };

In the above case, the prototype of display() function is the same in both the **base and derived class**. Therefore, the static binding cannot be applied. It would be great if the appropriate function is selected at the run time. This is known as **run time polymorphism**.

- **Run time polymorphism**: Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

Differences b/w compile time and run time polymorphism.

Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.

Unit 4 - Pointers:

Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

Unit 4 - Pointers:

C++ Runtime Polymorphism Example

Let's see a simple example of run time polymorphism in C++.

// an example without the virtual keyword.

```
#include <iostream>
using namespace std;
class Animal {
    public:
void eat(){
cout<<"Eating...";
    }
};
class Dog: public Animal
{
 public:
 void eat()
    {           cout<<"Eating bread...";
    }
};
int main(void) {
   Dog d = Dog();
   d.eat();
   return 0;
}
```

Output:

```
Eating bread...
```

C++ Run time Polymorphism Example: By using two derived class

Let's see another example of run time polymorphism in C++ where we are having two derived classes.

// an example with virtual keyword.

```
#include <iostream>
using namespace std;
class Shape {                              //  base class
    public:
virtual void draw(){                       // virtual function
cout<<"drawing..."<<endl;
```

Unit 4 - Pointers:

```
    }
};
class Rectangle: public Shape                 //  inheriting Shape class.
{
public:
void draw()
  {
      cout<<"drawing rectangle..."<<endl;
   }
};
class Circle: public Shape                    //  inheriting Shape class.

{
public:
void draw()
  {
     cout<<"drawing circle..."<<endl;
  }
};
int main(void) {
   Shape *s;                        //  base class pointer.
   Shape sh;                        // base class object.
      Rectangle rec;
       Circle cir;
     s=&sh;
    s->draw();
       s=&rec;
    s->draw();
   s=?
   s->draw();
}
```

Output:

```
drawing...
drawing rectangle...
drawing circle...
```

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

Unit 4 - Pointers:

What is an abstract class in C++?

By definition, a C++ abstract class must include at least one pure virtual function. Alternatively, put a function without a definition. Because the subclass would otherwise turn into an abstract class in and of itself, the abstract class's descendants must specify the pure virtual function.

Broad notions are expressed using abstract classes, which can then be utilized to construct more specific classes. You cannot make an object of the abstract class type. However, pointers and references can be used to abstract class types. When developing an abstract class, define at least one pure virtual feature. A virtual function is declared using the pure specifier (= 0) syntax.

Consider the example of the virtual function. Although the class's objective is to provide basic functionality for shapes, elements of type shapes are far too general to be of much value. Because of this, the shape is a good candidate for an abstract class:

Code

```
C-lass classname //abstract class
{
//data members
public:
//pure virtual function
/* Other members */
};
```

What are the characteristics of abstract class?

Although the Abstract class type cannot be created from scratch, it can have pointers and references made to it. A pure virtual function can exist in an abstract class in addition to regular functions and variables. Upcasting, which lets derived classes access their interface, is the main usage of abstract classes. Classes that descended from an abstract class must implement all pure virtues.

ADVERTISEMENT

ADVERTISEMENT

What are the restrictions to abstract class?

The following uses of abstract classes are not permitted:

Unit 4 - Pointers:

1. **Conversions made consciously**
2. **Member data or variables**
3. **Types of function output**
4. **Forms of debate**

It is unknown what happens when a pure virtual method is called explicitly or indirectly by the native code function Object () of an abstract class. Conversely, abstract group constructors and destructors can call additional member functions.

Although the constructors of the abstract class are allowed to call other member functions, if they either directly or indirectly call a pure virtual function, the outcome is unknown. But hold on! What exactly is a pure virtual function?

Let's first examine virtual functions in order to comprehend the pure virtual function.

A member function that has been redefined by a derived class from a base class declaration is referred to as a virtual function.

A virtual function that lacks definition or logic is known as an abstract function or a pure virtual function. At the time of declaration, 0 is assigned to it.

How Important Abstraction Is in Daily Life?

The ATM machine is another example of abstraction in everyday life;. However, we all use the ATM to do tasks like cash withdrawal, money transfers, generating mini-statements, and so on, and we have no access to the ATM's internal data. Data protection techniques like data abstraction can prevent unauthorized access to data.

What is the difference between abstract class and interface?

Interface	**Abstract class**
An interface can only inherit from another interface.	With the Extended keyword, an abstract class can enforce an interface and inherit from another class.
Use of the implements keyword is required in order to implement an interface.	Use the extends keyword to inherit from an abstract class.

What is an example of an abstract class?

Consider developing a calculator that will output the shape's perimeter when it is entered. Consider the type of programming you would use to create such a calculator. By creating distinct functions inside the Shape class, you may start with a few basic forms and hardcode the perimeter.

This is how the class might appear:

What is an example of an abstract class?

Consider developing a calculator that will output the shape's perimeter when it is entered. Consider the type of programming you would use to create such a calculator. By creating distinct functions inside the Shape class, you may start with a few basic forms and hardcode the perimeter.

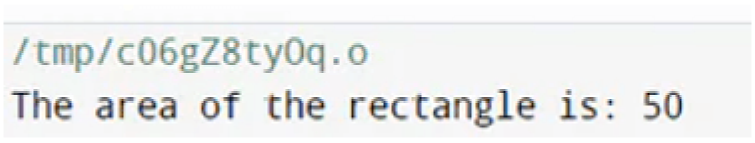
This is how the class might appear:

```
class Shape {
  public:
// All the functions of both square and rectangle are clubbed together in a single class.
void width(int w) {
shape_width = w;
}
void height(int h) {
shape_height = h;
}
int areaOfSquare(int s) {
return 4 * s;
}
int areaOfRectange(int l, int b) {
return (l * b);
}
protected:
int shape_width;
int shape_height;
};
```

Unit 4 - Pointers:

```
int main (){
shapes R;
R.width(5);
R.height(10);
cout<<"The area of rectangle is"<<R.areaOfRectangle";
return 0;
}
```

Output:



```
/tmp/c06gZ8ty0q.o
The area of the rectangle is: 50
```