OBJECT PROGRAMMING

- LECTURE 9 -

(1ST YEAR OF STUDY)

Skopje, 2023/24

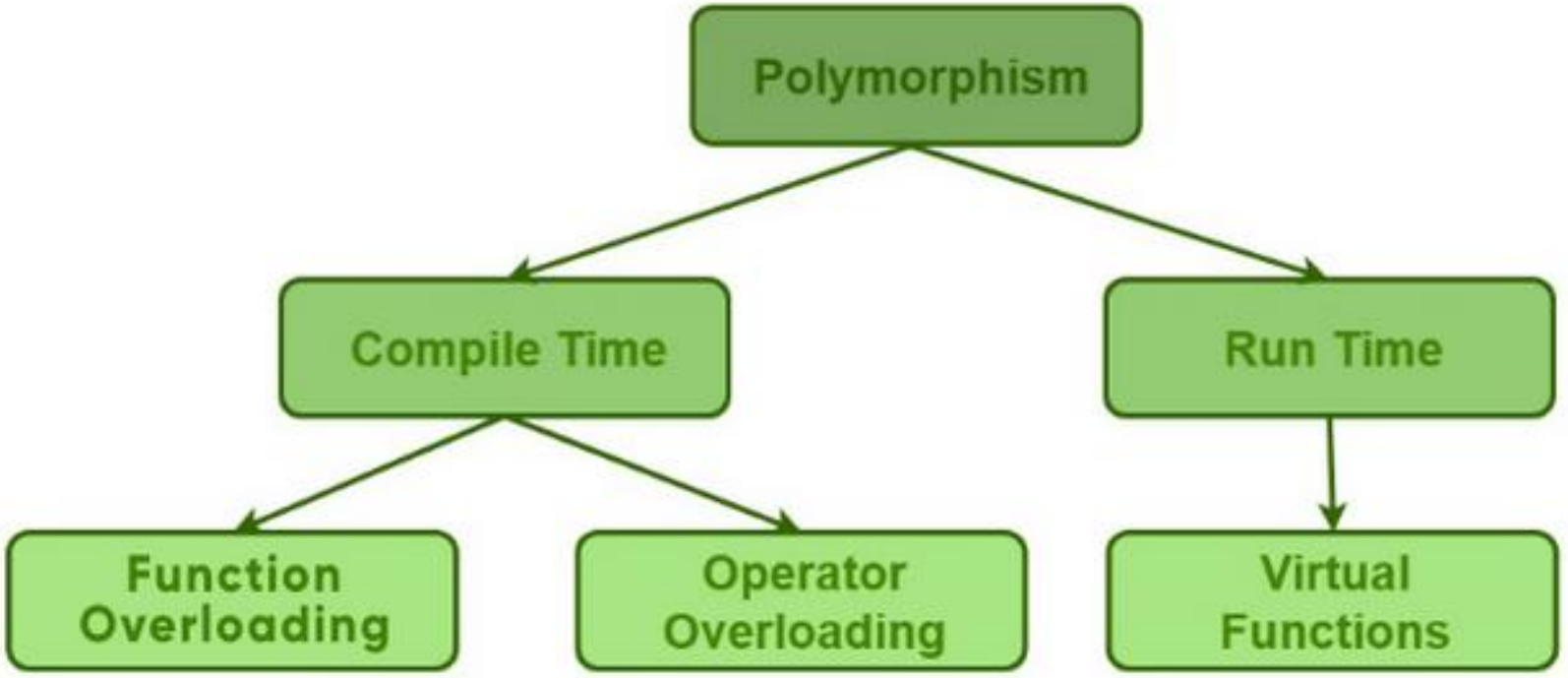
Contents

9.1. Introduction

- The word "***Polymorphism***" means having many forms.
- In simple words, we can define *polymorphism* as the ability of a message to be displayed in more than one form.
- A real-life example of polymorphism is a person who at the same time can have different characteristics. So the same person exhibits different behavior in different situations. This is called polymorphism. Polymorphism is considered one of the important features of OOP.

9.1. Introduction

- **Types of Polymorphism**
  - Compile-time Polymorphism
  - Run-time Polymorphism



9.1. Introduction

- **1. Compile-Time Polymorphism**
  - This type of polymorphism is achieved by **Function overloading** or **Operator overloading**.
- **2. Run-time Polymorphism**
  - This type of polymorphism is achieved by **Function overriding**. *Late binding* and *Dynamic polymorphism* are other names for run-time polymorphism. The function call is resolved at run-time in ***run-time polymorphism***.
  - In contrast, with compile-time polymorphism, the compiler determines which function call to bind to the object after deducing it at run-time.

9.1. Introduction

- **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*.
- Run-time Polymorphism cannot be achieved by ***data members*** in C++.
- Let's see an example where we are accessing the field by reference variable of *parent class*, which refers to the instance of the *derived class*.
- We can see that the parent class reference will always refer to the data member of the parent class.

9.1. Introduction


```cpp
// C++ program for function overriding with data members
#include <bits/stdc++.h>
using namespace std;

//  base class declaration.
class Animal {
public:
    string color = "Black";
};

// inheriting Animal class.
class Dog : public Animal {
public:
    string color = "Grey";
};

// Driver code
int main(void)
{
    Animal d = Dog(); // accessing the field by reference
                      // variable which refers to derived
    cout << d.color;
}
```

Output

```
Black
```

9.1. Introduction

- A ***pointer to a base class*** CAN point to an ***object of a derived class***. Using the pointer one can reach only the members of the derived class that are inherited from the base class.
  - This is due to the fact that the pointer knows only the features of the base class.
  - The pointer has no knowledge of the number and the type of members added by the derived classes.

9.1. Introduction

- A ***pointer to a derived class*** CANNOT point to an ***object of a base class***. Since the derived class will always have more features than the base class, and a pointer to that type will need to have access to all of those features.
  - At the very least a derived class will define a (default) constructor to the derived class type – something which the base class will not have access to.

9.1. Introduction

- A **Virtual function** is a member function that is declared in the *base class* using the keyword ***virtual*** and is re-defined (Overridden) in the derived class.
- Some key points about Virtual functions:
  - Virtual functions are Dynamic in nature.
  - They are defined by inserting the keyword "**virtual**" inside a base class and are always declared with a base class and overridden in a child class.
  - A virtual function is called during Runtime.
- Below is the C++ program to demonstrate *virtual function*:

9.1. Introduction


```cpp
// C++ Program to demonstrate
// the Virtual Function
#include <iostream>
using namespace std;

// Declaring a Base class
class GFG_Base {

public:
    // virtual function
    virtual void display()
    {
        cout << "Called virtual Base Class function"
             << "\n\n";
    }

    void print()
    {
        cout << "Called GFG_Base print function"
             << "\n\n";
    }
};
```

Output

```
Called virtual Base Class function

Called GFG_Base print function
```

```cpp
// Declaring a Child Class
class GFG_Child : public GFG_Base {

public:
    void display()
    {
        cout << "Called GFG_Child Display Function"
             << "\n\n";
    }

    void print()
    {
        cout << "Called GFG_Child print Function"
             << "\n\n";
    }
};

    // Driver code
    int main()
    {
        // Create a reference of class GFG_Base
        GFG_Base* base;

        GFG_Child child;

        base = &child;

        // This will call the virtual function
        base->GFG_Base::display();

        // this will call the non-virtual function
        base->print();
    }
```

9.1. Introduction


```cpp
// C++ program for virtual function overriding
#include <bits/stdc++.h>
using namespace std;

class base {
public:
    virtual void print()
    {
        cout << "print base class" << endl;
    }

    void show() { cout << "show base class" << endl; }
};

class derived : public base {
public:
    // print () is already virtual function in
    // derived class, we could also declared as
    // virtual void print () explicitly
    void print() { cout << "print derived class" << endl; }

    void show() { cout << "show derived class" << endl; }
};
```

```cpp
// Driver code
int main()
{
    base* bptr;
    derived d;
    bptr = &d;

    // Virtual function, binded at
    // runtime (Runtime polymorphism)
    bptr->print();

    // Non-virtual function, binded
    // at compile time
    bptr->show();

    return 0;
}
```

Output

```
print derived class
show base class
```

9.1. Introduction

- A ***function*** is a block of statements that together performs a specific task by taking some input and producing a particular output.
- **Function overriding in C++** is termed as the redefinition of base class function in its derived class with the same signature, i.e., ***return type*** and ***parameters***.
- It falls under the category of *Run-time Polymorphism*.

9.1. Introduction


```
// C++ program to demonstrate function overriding

#include <iostream>
using namespace std;

class Parent {
public:
    void GeeksforGeeks_Print()
    {
        cout << "Base Function" << endl;
    }
};

class Child : public Parent {
public:
    void GeeksforGeeks_Print()
    {
        cout << "Derived Function" << endl;
    }
};

int main()
{
    Child Child_Derived;
    Child_Derived.GeeksforGeeks_Print();
    return 0;
}
```

Output

```
Derived Function
```

9.1. Introduction

- **1. Call Overridden Function From Derived Class**

```cpp
// C++ program to demonstrate function overriding
// by calling the overridden function
// of a member function from the child class

#include <iostream>
using namespace std;

class Parent {
public:
    void GeeksforGeeks_Print()
    {
        cout << "Base Function" << endl;
    }
};

class Child : public Parent {
public:
    void GeeksforGeeks_Print()
    {
        cout << "Derived Function" << endl;

        // call of overridden function
        Parent::GeeksforGeeks_Print();
    }
};
```

```cpp
int main()
{
    Child Child_Derived;
    Child_Derived.GeeksforGeeks_Print();
    return 0;
}
```

Output

```
Derived Function
Base Function
```

9.1. Introduction

- **2. Call Overridden Function Using Pointer**

```cpp
// C++ program to access overridden function using pointer
// of Base type that points to an object of Derived class
#include <iostream>
using namespace std;

class Parent {
public:
    void GeeksforGeeks()
    {
        cout << "Base Function" << endl;
    }
};

class Child : public Parent {
public:
    void GeeksforGeeks()
    {
        cout << "Derived Function" << endl;
    }
};

int main()
{
    Child Child_Derived;

    // pointer of Parent type that points to derived1
    Parent* ptr = &Child_Derived;

    // call function of Base class using ptr
    ptr->GeeksforGeeks();

    return 0;
}
```

Output

```
Base Function
```

9.1. Introduction

- **3. Access of Overridden Function to the Base Class**

```cpp
// C++ program to access overridden function
// in main() using the scope resolution operator ::

#include <iostream>
using namespace std;

class Parent {
public:
    void GeeksforGeeks()
    {
        cout << "Base Function" << endl;
    }
};

class Child : public Parent {
public:
    void GeeksforGeeks()
    {
        cout << "Derived Function" << endl;
    }
};

int main()
{
     Child Child_Derived;
     Child_Derived.GeeksforGeeks();

    // access GeeksforGeeks() function of the Base class
     Child_Derived.Parent::GeeksforGeeks();
    return 0;
}
```

Output

```
Derived Function
Base Function
```

9.1. Introduction

- **4. Access to Overridden Function**

```cpp
// C++ Program Demonstrating
// Accessing of Overridden Function
#include <iostream>
using namespace std;

// defining of the Parent class
class Parent

{

public:
    // defining the overridden function
    void GeeksforGeeks_Print()
    {
        cout << "I am the Parent class function" << endl;
    }
};
```

Output

```
I am the Child class function
I am the Parent class function
```

```cpp
// defining of the derived class
class Child : public Parent

{
public:
    // defining of the overriding function
    void GeeksforGeeks_Print()
    {
        cout << "I am the Child class function" << endl;
    }
};

int main()
{
    // create instances of the derived class
    Child GFG1, GFG2;

    // call the overriding function
    GFG1.GeeksforGeeks_Print();

    // call the overridden function of the Base class
    GFG2.Parent::GeeksforGeeks_Print();
    return 0;
}
```

9.1. Introduction

Function Overloading Vs Function Overriding

Function Overloading	Function Overriding
It falls under Compile-Time polymorphism	It falls under Runtime Polymorphism
A function can be overloaded multiple times as it is resolved at Compile time	A function cannot be overridden multiple times as it is resolved at Run time
Can be executed without inheritance	Cannot be executed without inheritance
They are in the same scope	They are of different scopes.

9.1. Introduction

- The idea is that *virtual functions* are called according to the ***type of the object instance pointed to or referenced***, not according to the ***type of the pointer or reference***. In other words, virtual functions are resolved late, at run-time.
- **What is the use?**
  - Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing the kind of derived class object.
- To calculate the *Area of Shapes* by *Virtual Function*.

9.1. Introduction


```
// C++ program to demonstrate how we will calculate
// the area of shapes USING VIRTUAL FUNCTION
#include <fstream>
#include <iostream>
using namespace std;

// Declaration of Base class
class Shape {
public:
    // Usage of virtual constructor
    virtual void calculate()
    {
        cout << "Area of your Shape ";
    }
    // usage of virtual Destuctor to avoid memory leak
    virtual ~Shape()
    {
        cout << "Shape Destuctor Call\n";
    }
};
```

```
// Declaration of Derived class
class Rectangle : public Shape {
public:
    int width, height, area;

    void calculate()
    {
        cout << "Enter Width of Rectangle: ";
        cin >> width;

        cout << "Enter Height of Rectangle: ";
        cin >> height;

        area = height * width;
        cout << "Area of Rectangle: " << area << "\n";
    }

    // Virtual Destuctor for every Derived class
    virtual ~Rectangle()
    {
        cout << "Rectangle Destuctor Call\n";
    }
};
```

9.1. Introduction


```
// Declaration of 2nd derived class
class Square : public Shape {
public:
    int side, area;

    void calculate()
    {
        cout << "Enter one side your of Square: ";
        cin >> side;

        area = side * side;
        cout << "Area of Square: " << area << "\n";
    }

   // Virtual Destuctor for every Derived class
    virtual ~Square()
    {
        cout << "Square Destuctor Call\n";
    }
};
```

Output:

```
Enter Width of Rectangle: 10
Enter Height of Rectangle: 20
Area of Rectangle: 200
Enter one side your of Square: 16
Area of Square: 256
```

```
int main()
{

    // base class pointer
    Shape* S;
    Rectangle r;

    // initialization of reference variable
    S = &r;

    // calling of Rectangle function
    S->calculate();
    Square sq;

    // initialization of reference variable
    S = &sq;

    // calling of Square function
    S->calculate();

    // return 0 to tell the program executed
    // successfully
    return 0;
}
```

9.1. Introduction

- Real-Life Example to Understand the Implementation of Virtual Function
  - Consider ***employee management software*** for an organization.
  - Let the code has a simple base class *Employee*, the class contains *virtual functions* like ***raiseSalary()***, ***transfer()***, ***promote()***, etc.
  - Different types of employees like *Managers*, *Engineers*, etc., may have their own implementations of the virtual functions present in base class *Employee*.

9.1. Introduction

- In our complete software, we just need to *pass* a list of employees everywhere, and *call* appropriate functions without even knowing the type of employee.
- For example, we can easily *raise the salary* of all employees by iterating through the list of employees.
- Every type of employee may have its own logic in its class, but we don't need to worry about them because if *raiseSalary()* is present for a specific employee type, only that function would be called.

9.1. Introduction


```
// C++ program to demonstrate how a virtual function
// is used in a real life scenario

class Employee {
public:
    virtual void raiseSalary()
    {
        // common raise salary code
    }

    virtual void promote()
    {
        // common promote code
    }
};

class Manager : public Employee {
    virtual void raiseSalary()
    {
        // Manager specific raise salary code, may contain
        // increment of manager specific incentives
    }

    virtual void promote()
    {
        // Manager specific promote
    }
};
```

```
// Similarly, there may be other types of employees

// We need a very simple function
// to increment the salary of all employees
// Note that emp[] is an array of pointers
// and actual pointed objects can
// be any type of employees.
// This function should ideally
// be in a class like Organization,
// we have made it global to keep things simple
void globalRaiseSalary(Employee* emp[], int n)
{
    for (int i = 0; i < n; i++) {
        // Polymorphic Call: Calls raiseSalary()
        // according to the actual object, not
        // according to the type of pointer
        emp[i]->raiseSalary();
    }
}
```

9.2. Inheritance vs Polymorphism

- **Inheritance** is one in which a new class is created that inherits the properties of the already existing class. It supports the concept of code reusability and reduces the length of the code in OPP.
  - For example, class B is the derived class which inherit the property (i.e., method) of the base class A.
- **Polymorphism** is that in which we can perform a task in multiple forms or ways. It is applied to the *functions* or *methods*. Polymorphism allows the object to decide which form of the function to implement at compile-time as well as run-time.

9.2. Inheritance vs Polymorphism

- Types of Polymorphism are:
  - Compile-time polymorphism (Method Overloading)
  - Run-time polymorphism (Method Overriding)

```cpp
#include "iostream"
using namespace std;

class A {
    int a, b, c;

public:
    void add(int x, int y)
    {
        a = x;
        b = y;
        cout << "addition of a+b is:" << (a + b) << endl;
    }

    void add(int x, int y, int z)
    {
        a = x;
        b = y;
        c = z;
        cout << "addition of a+b+c is:" << (a + b + c) << endl;
    }

    virtual void print()
    {
        cout << "Class A's method is running" << endl;
    }
};
```

```cpp
class B : public A {
public:
    void print()
    {
        cout << "Class B's method is running" << endl;
    }
};

int main()
{
    A a1;

    // method overloading (Compile-time polymorphism)
    a1.add(6, 5);

    // method overloading (Compile-time polymorphism)
    a1.add(1, 2, 3);

    B b1;

    // Method overriding (Run-time polymorphism)
    b1.print();
}
```

Output

```
addition of a+b is:11
addition of a+b+c is:6
Class B's method is running
```

9.2. Inheritance vs Polymorphism

- A potential problem arises when multiple base classes are inherited by a derived class.
  - In this case it will be unknown, e.g., whether C will inherit its members through the B1 class, or the B2 class.
- This problem is resolved by utilizing **Virtual Inheritance**.
  - It is implemented by inserting the keyword ***virtual*** prior to the *access modifier* of the inheritance of the base class.

9.2. Inheritance vs Polymorphism

- When utilizing *standard inheritance*, copies of all superclasses of the superclass are created.
  - This would then create ambiguity, depending on which superclass instance to refer to.
- When utilizing *virtual inheritance*, only one copy of the superclasses is created.
  - Thus there is no ambiguity as to which instance to invoke.
  - This case is handled individually by C++.

9.2. Inheritance vs Polymorphism


```
#include <iostream>
using namespace std;
class base{
   public:
        int i;
};

class derived1: virtual public base{
   public:
        int j;
};

class derived2: virtual public base{
   public:
        int k;
};

class derived3: public derived2, public derived1 {
   public:
        int mult() { return i*j*k; }
};
```

```
int main()
{
        derived3 ob;
        ob.i=10;
        ob.j=3;
        ob.k=5;
        cout<<"Result of multiplication: " <<ob.mult()<<endl;
        return 0;
}
```

```
Result of multiplication: 150
```

- If *derived1* and *derived2* did not inherit *base* as a virtual class, then the statement ob.i=10; would give an ambiguity error in compile time.

9.3. Virtual Functions

- A **Virtual function** (also known as **Virtual method**) is a member function that is declared within a base class and is re-defined (overridden) by a derived class.
- 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 method.
- **Note:** Never call a virtual function from a **constructor** or **destructor**.

9.3. Virtual Functions

- ***Virtual functions*** ensure that the correct function is called for an object, *regardless* of the type of reference (or pointer) used for the function call.
- They are mainly used to achieve *Run-time polymorphism*.
- Functions are declared with a **virtual** keyword in a base class. The resolving of a function call is done at run-time.
- **Note:** If we have created a virtual function in the base class and it is being overridden in the derived class, then we don't need a virtual keyword in the derived class, functions are automatically considered virtual functions in the derived class.

9.3. Virtual Functions

- The **rules for the virtual functions** in C++ are as follows:
  - Virtual functions cannot be static.
  - A virtual function can be a friend function of another class.
  - Virtual functions should be accessed using a pointer or reference of base class type to achieve run-time polymorphism.
  - The prototype of virtual functions should be the same in the base as well as the derived class.
  - They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
  - A class may have a ***virtual destructor***, but it cannot have a virtual constructor.

9.3. Virtual Functions

- Limitations of Virtual Functions:
  - **Slower:** The function call takes slightly longer due to the virtual mechanism and makes it more difficult for the compiler to optimize because it does not know exactly which function is going to be called at compile-time.
  - **Difficult to Debug:** In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from.

9.3. Virtual Functions

- To see the combined application of ***default arguments*** with ***virtual functions*** let's take a sample.

```
// C++ program To demonstrate how default arguments
// and virtual function are used together
#include <iostream>
using namespace std;

class Base {
public:
    virtual void fun(int x = 0)
    {
        cout << "Base::fun(), x = " << x << endl;
    }
};

class Derived : public Base {
public:
    virtual void fun(int x = 10) // NOTE THIS CHANGE
    {
        cout << "Derived::fun(), x = " << x << endl;
    }
};

int main()
{
    Derived d1; // Constructor

    // Base class pointer which will
    // Edit value in memory location of
    // Derived class constructor
    Base* bp = &d1;

    bp->fun(); // Calling a derived class member function

    return 0; // Returning 0 means the program
              // Executed successfully
}
```

Output

```
Derived::fun(), x = 0
```

9.3. Virtual Functions

- If we take a closer look at the output, we observe that the '*fun()*' function of the derived class is called, and the ***default value*** of the base class '*fun()*' function is used.
- Default arguments do not participate in the signature (function's name, type, and order) of functions.
- So signatures of the '*fun()*' function in the base class and derived class are considered the *same*, hence the '*fun()*' function of the base class is **overridden**.
- Also, the default value is used at compile time. When the compiler sees an argument missing in a function call, it substitutes the default value given.

9.3. Virtual Functions

- Therefore, in the above program, the value of **x** is substituted at compile-time, and at run-time derived class's '*fun()*' function is called.
- Thus, the default value is substituted at compile-time. The fun() is called on 'bp' which is a pointer of Base type. So, compiler substitutes 0 (not 10).

9.3. Virtual Functions

- Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a **Virtual Destructor**.
- As a guideline, any time you have a virtual function in a class, you should immediately add a virtual destructor (even if it does nothing). This way, you ensure against any surprises later.
- Making base class destructor *virtual* guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called.

9.3. Virtual Functions


```
// A program with virtual destructor
#include <iostream>

using namespace std;

class base {
  public:
    base()
    { cout << "Constructing base\n"; }
    virtual ~base()
    { cout << "Destructing base\n"; }
};

class derived : public base {
  public:
    derived()
    { cout << "Constructing derived\n"; }
    ~derived()
    { cout << "Destructing derived\n"; }
};
```

```
int main()
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

Output

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

9.3. Virtual Functions

- Can we make a class constructor *virtual* in C++ to create polymorphic objects? No.
- C++ being a statically typed language, it is meaningless to the C++ compiler to create an object polymorphically. The compiler must be aware of the class type to create the object.
- In other words, what type of object to be created is a compile-time decision from the C++ compiler perspective. If we make a constructor virtual, the compiler flags an error.

9.3. Virtual Functions

- Sometimes implementation of all functions cannot be provided in a base class because we don't know the implementation. Such a class is called an **Abstract Class**.
- For example, let ***Shape*** be a base class. We cannot provide the implementation of function draw() in Shape, but we know every derived class must have an implementation of draw().
- Similarly, an ***Animal*** class doesn't have the implementation of move() (assuming that all animals move), but all animals must know how to move.
- We cannot create objects of abstract classes.

9.3. Virtual Functions

- A **Pure virtual function** (or abstract function) in C++ is a *virtual function* for which we can have an implementation, but we must override that function in the derived class, otherwise, the derived class will also become an abstract class.
- A pure virtual function is declared by assigning 0 in the declaration.
- A pure virtual function is implemented by classes that are derived from an Abstract class.

9.3. Virtual Functions


```cpp
// C++ Program to illustrate the abstract class and virtual
// functions
#include <iostream>
using namespace std;

class Base {
    // private member variable
    int x;

public:
    // pure virtual function
    virtual void fun() = 0;

    // getter function to access x
    int getX() { return x; }
};

// This class inherits from Base and implements fun()
class Derived : public Base {
    // private member variable
    int y;

public:
    // implementation of the pure virtual function
    void fun() { cout << "fun() called"; }
};
```

```cpp
int main(void)
{
    // creating an object of Derived class
    Derived d;

    // calling the fun() function of Derived class
    d.fun();

    return 0;
}
```

Output

```
fun() called
```

9.3. Virtual Functions

- Some Interesting Facts:
  - 1. A class is abstract if it has at least one pure virtual function.
  - 2. We can have pointers and references of abstract class type.
  - 3. If we do not override the pure virtual function in the derived class, then the derived class also becomes an abstract class.
  - 4. An abstract class can have constructors.
  - 5. An abstract class in C++ can also be defined using struct keyword.

9.3. Virtual Functions

- An **Interface** does not have an implementation of any of its methods, it can be considered as a collection of method declarations. In C++, an interface can be simulated by making all methods pure virtual. In Java, there is a separate keyword for the interface.
- We can think of Interface as header files in C++, like in header files we only provide the body of the class that is going to implement it. Similarly, in Java in Interface we only provide the body of the class and we write the actual code in whatever class implements it.

9.3. Virtual Functions

- **Note:** Only Destructors can be Virtual. Constructors cannot be declared as virtual, this is because if you try to override a constructor by declaring it in a base/super class and call it in the derived/sub class with same functionalities it will always give an error as overriding means a feature that lets us to use a method from the parent class in the child class which is not possible.
- It is possible to have a ***Pure virtual destructor*** in C++. They are legal and one of the most important things to remember is that if a class contains a pure virtual destructor, it must provide a function body for the pure virtual destructor.

9.3. Virtual Functions

- In C++, a Static member function of a class cannot be virtual.
- Virtual functions are invoked when you have a pointer or reference to an instance of a class.
- Static functions aren't tied to the instance of a class but they are tied to the class.
- C++ doesn't have pointers-to-class, so there is no scenario in which you could invoke a static function virtually.

9.3. Virtual Functions

- The **Run-time Cast,** which checks that the cast is valid, is the simplest approach to ascertain the run-time type of an object using a pointer or reference.
- This is especially beneficial when we need to cast a pointer from a base class to a derived type. When dealing with the inheritance hierarchy of classes, the casting of an object is usually required. There are two types of casting:
  - **Upcasting:** When a pointer or a reference of a derived class object is treated as a base class pointer.
  - **Downcasting:** When a base class pointer or reference is converted to a derived class pointer.