CC-211L

Object Oriented Programming

Laboratory 11

Polymorphism

Version: 1.0.0

Release Date: 06-04-2023

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

Learning Objectives:

- Function Overriding
- Virtual Functions
- Virtual Destructors
- Pure Virtual Functions
- Abstract Classes

Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

General Instructions:

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how is s/he doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the followings.

Teachers:		
Course Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Lab Instructor	Azka Saddiqa	azka.saddiqa@pucit.edu.pk
Teacher Assistants	Saad Rahman	bsef19m021@pucit.edu.pk
	Zain Ali Shan	bcsf19a022@pucit.edu.pk

Background and Overview:

Polymorphism:

Polymorphism causes a member function to behave differently based on the object that calls/invokes it. Polymorphism is a Greek word that means to have many forms. It occurs when you have a hierarchy of classes related through inheritance.

Virtual Function:

A virtual function is a base class member function that you can redefine in a derived class to achieve polymorphism. You can declare the function in the base class using the virtual keyword.

Virtual Destructor:

A member function that is used to free up the memory allocated used by the object of a child class or derived class when it is removed from the memory using the parent class pointer object. Virtual destructor is mainly responsible for resolving the memory leak problem.

Overriding:

Function overriding is the redefinition of a base class function in its derived class with the same signature.

Abstract Class:

A class that is designed to be specifically used as a base class. An abstract class contains at least one pure virtual function.

Activities:

Pre-Lab Activities:

Function Overriding:

The function overriding in C++ is used to achieve runtime polymorphism. Runtime polymorphism is also known as dynamic polymorphism or late binding. In runtime polymorphism, the function call is resolved at run time. In contrast, to compile-time or static polymorphism, the compiler deduces the object at run time and then decides which function call to bind to the object. Following is the syntax of function overriding:

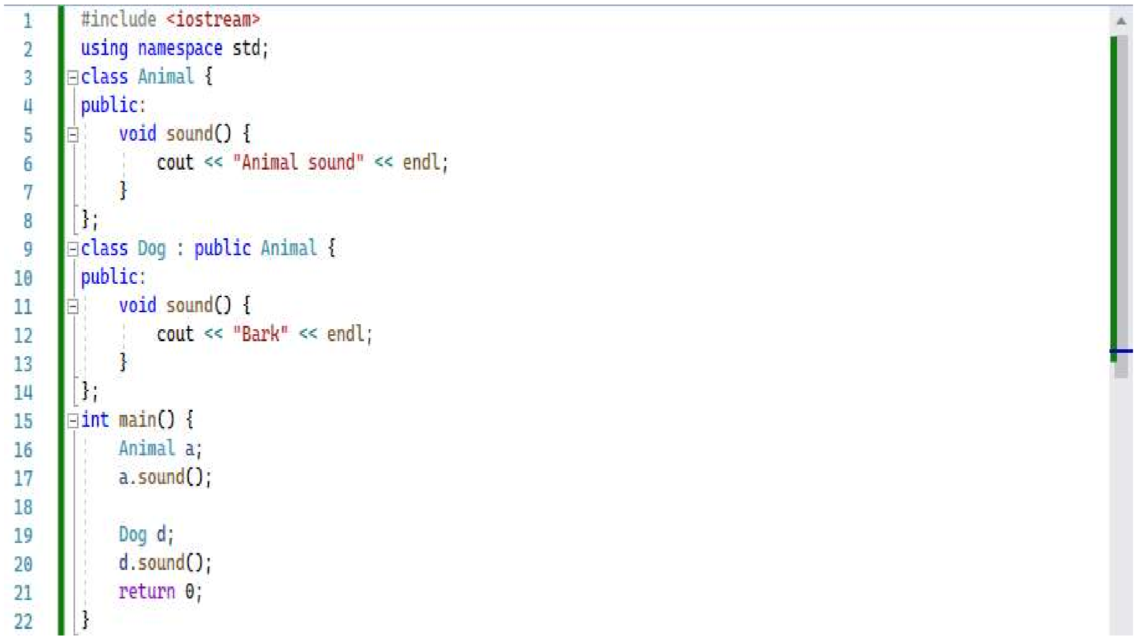
```
class Parent
{
  public:
   func()
  {
          //function definition
   }
};
class Child : public Parent
{
  public:
   func()
  {
          //function definition
   }
};
```

The Child class inherits the Parent class thus the func also gets inherited. After that func with a new definition is created within the child.

Imagine you are supposed to make classes for two different models of a car. Since both the classes will have a lot of features in common, you have decided to make a parent class Car and inherit those properties into your individual classes.

Now suppose you have two classes Volkswagen and Mercedes that inherit the Car class. The Car class has a function that displays the color of the Car (which by default is white). Now in your individual classes, you have defined methods to get the colorName. Thus, in this case, we will override the displayColor function to display the color input by the user.

Example:

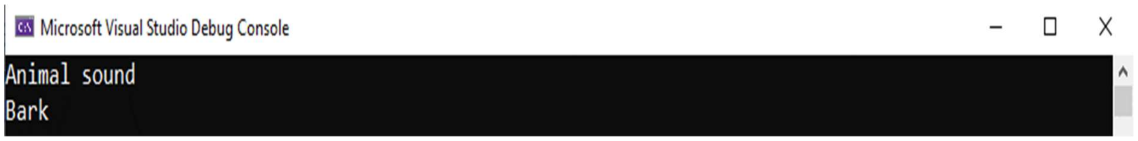


```cpp
#include <iostream>
using namespace std;
class Animal {
public:
    void sound() {
        cout << "Animal sound" << endl;
    }
};
class Dog : public Animal {
public:
    void sound() {
        cout << "Bark" << endl;
    }
};
int main() {
    Animal a;
    a.sound();

    Dog d;
    d.sound();
    return 0;
}
```

Fig. 01 (Function Overriding)

Output:



Microsoft Visual Studio Debug Console

```
Animal sound
Bark
```

Fig. 02 (Function Overriding)

Task 01: Calculate Area **[Estimated time 20 minutes / 10 marks]**

- Create a base class called Shape with a public member function area() that returns 0
- Then, create two subclasses called Rectangle and Triangle that inherit from the Shape class
- Override the area() function in both subclasses to calculate and return the area of a rectangle and a triangle, respectively.
- Finally, create instances of both the Rectangle and Triangle classes, and call the area() function on each instance to verify that the correct area is calculated and returned for each shape.

Task 02: Animal Sound **[Estimated time 20 minutes / 10 marks]**

- Create a base class called Animal with a public member function called make_sound() that prints "The animal makes a sound."
- Create three subclasses called Cat, Dog, and Bird that inherit from the Animal class.
- Override the make_sound() function in each subclass to print "Meow" for Cat, "Woof" for Dog, and "Chirp" for Bird.
- Finally, create an array of pointers to Animal objects that contains one instance of each subclass, and call the make_sound() function on each object to verify that the correct sound is made for each animal.

In-Lab Activities:

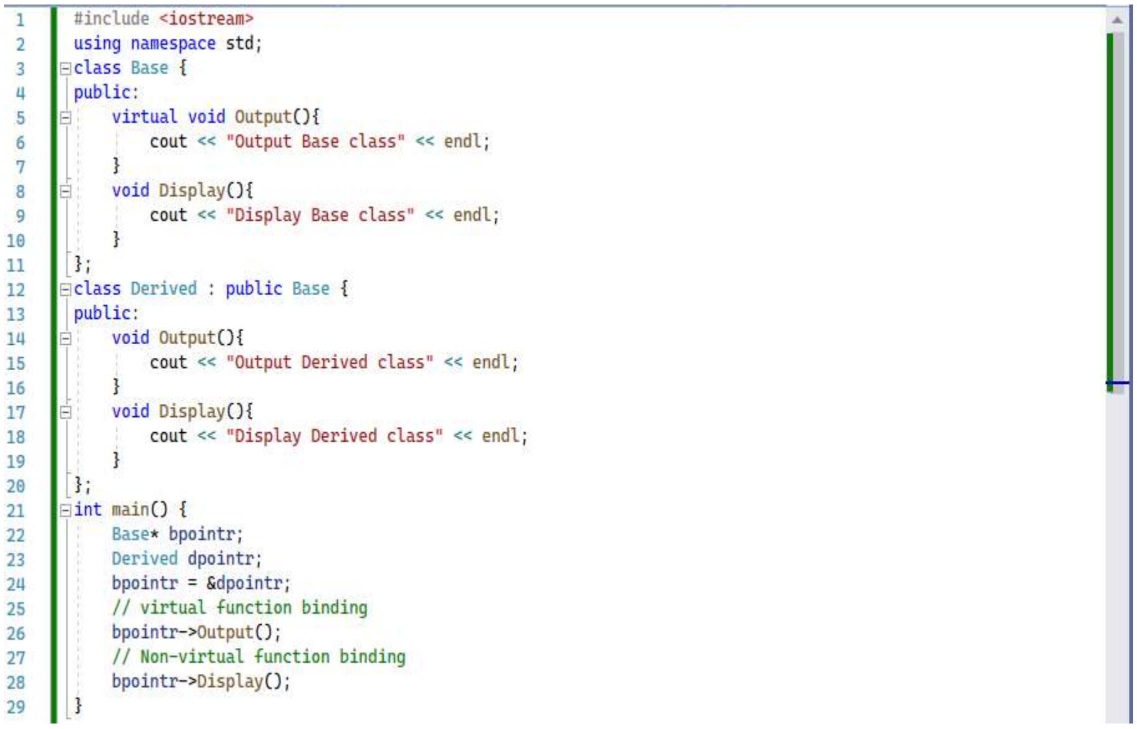
Virtual Function:

A virtual function is a base class member function that you can redefine in a derived class to achieve polymorphism. You can declare the function in the base class using the virtual keyword. Once you declare the function in the base class, you can use a pointer or reference to call the virtual class and execute its virtual version in the derived class. Thus, it asks the compiler to determine the object's type during run-time and create a function bind.

Rules of Virtual Function:

- The functions cannot be static
- You derive them using the "virtual" keyword
- Virtual functions in C++ needs to be a member of some other class (base class)
- They can be a friend function of another class
- The prototype of these functions should be the same for both the base and derived class
- Virtual functions are accessible using object pointers
- Redefining the virtual function in the derived class is optional, but it needs to be defined in the base class
- The function call resolving is done at run-time
- You can create a virtual destructor but not a constructor

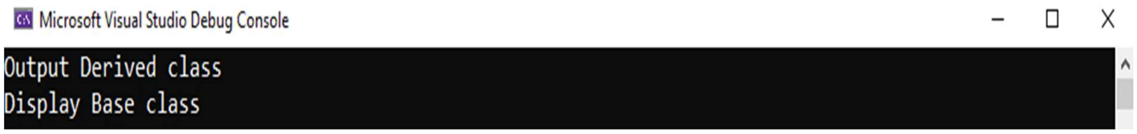
Example:



```cpp
#include <iostream>
using namespace std;
class Base {
public:
    virtual void Output(){
        cout << "Output Base class" << endl;
    }
    void Display(){
        cout << "Display Base class" << endl;
    }
};
class Derived : public Base {
public:
    void Output(){
        cout << "Output Derived class" << endl;
    }
    void Display(){
        cout << "Display Derived class" << endl;
    }
};
int main() {
    Base* bpointr;
    Derived dpointr;
    bpointr = &dpointr;
    // virtual function binding
    bpointr->Output();
    // Non-virtual function binding
    bpointr->Display();
}
```

Fig. 03 (Virtual Function)

Output:



Microsoft Visual Studio Debug Console

```
Output Derived class
Display Base class
```

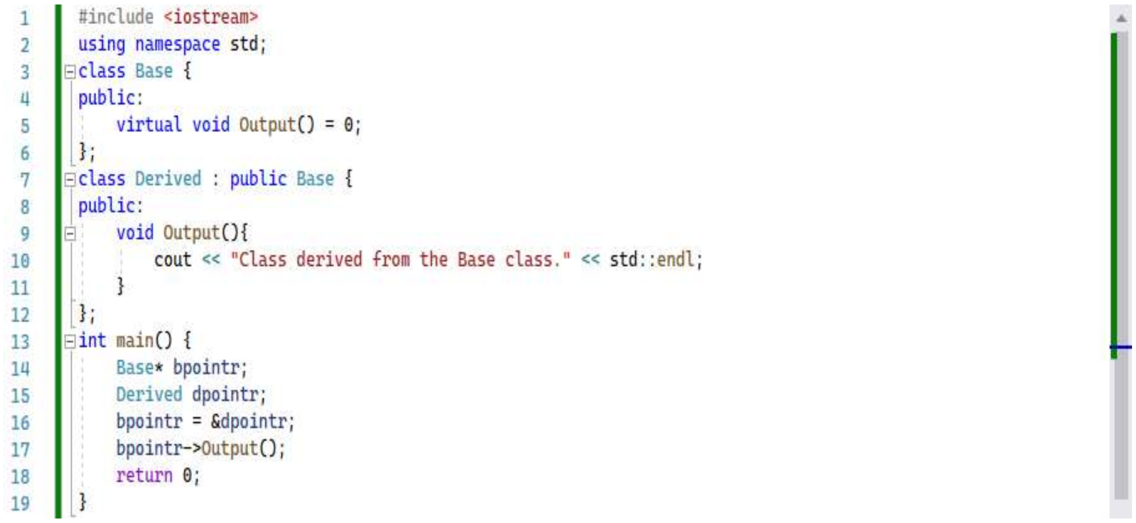
Fig. 04 (Virtual Function)

Pure Virtual Function:

A pure virtual function is a virtual function that does not perform any task. It is only used as a placeholder and does not contain any function definition. It is declared in an abstract base class. You derive a pure virtual function in C++ using the following syntax:

virtual void function_name() = 0;

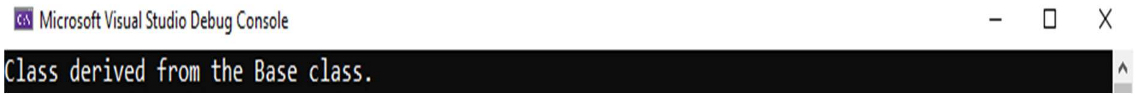
Example:



```cpp
#include <iostream>
using namespace std;
class Base {
public:
    virtual void Output() = 0;
};
class Derived : public Base {
public:
    void Output(){
        cout << "Class derived from the Base class." << std::endl;
    }
};
int main() {
    Base* bpointr;
    Derived dpointr;
    bpointr = &dpointr;
    bpointr->Output();
    return 0;
}
```

Fig. 05 (Pure Virtual Function)

Output:



Microsoft Visual Studio Debug Console

```
Class derived from the Base class.
```

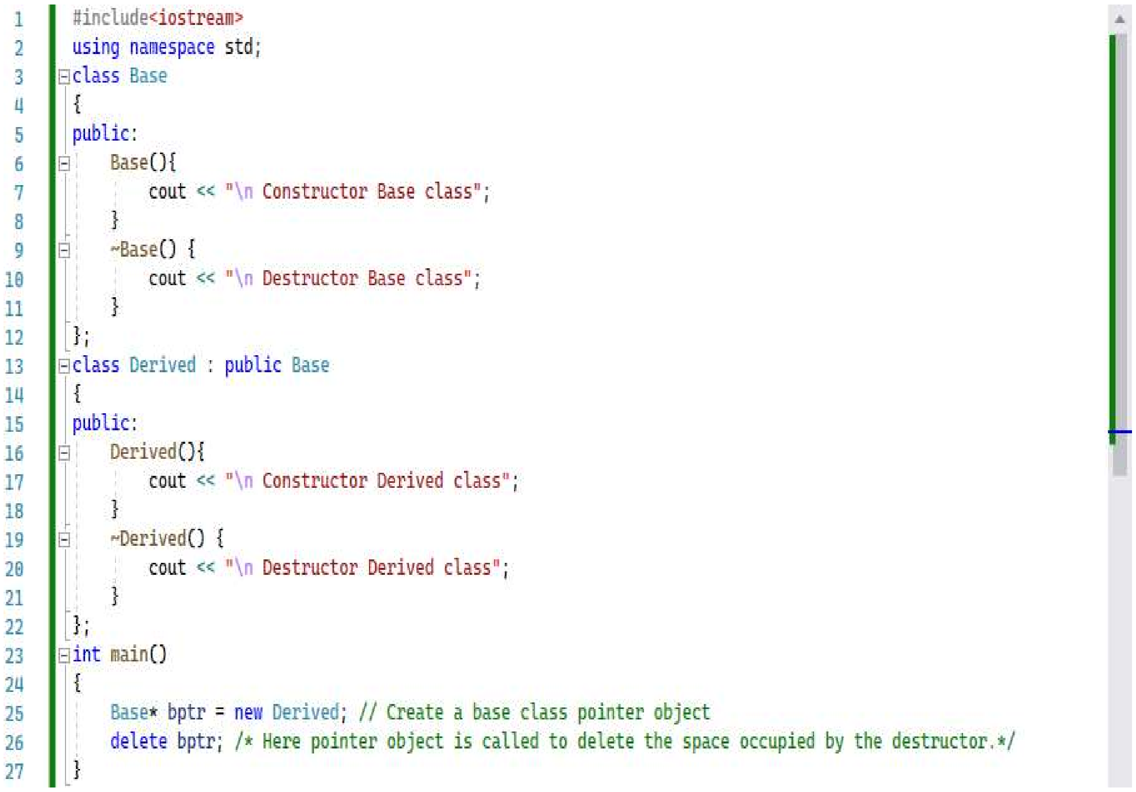
Fig. 06 (Pure Virtual Function)

Virtual Destructor:

A virtual destructor is used to free up the memory space allocated by the derived class object or instance while deleting instances of the derived class using a base class pointer object. A base or parent class destructor use the virtual keyword that ensures both base class and the derived class destructor will be called at run time, but it called the derived class first and then base class to release the space occupied by both destructors.

Following examples demonstrates the behavior with and without the virtual destructor.

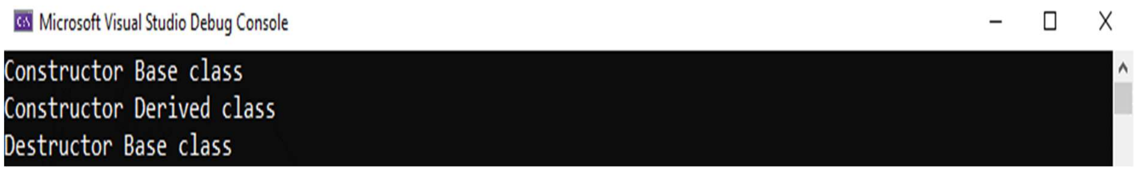
Example:



```cpp
#include<iostream>
using namespace std;
class Base
{
public:
    Base(){
        cout << "\n Constructor Base class";
    }
    ~Base() {
        cout << "\n Destructor Base class";
    }
};
class Derived : public Base
{
public:
    Derived(){
        cout << "\n Constructor Derived class";
    }
    ~Derived() {
        cout << "\n Destructor Derived class";
    }
};
int main()
{
    Base* bptr = new Derived; // Create a base class pointer object
    delete bptr; /* Here pointer object is called to delete the space occupied by the destructor.*/
}
```

Fig. 07 (without using Virtual Destructor)

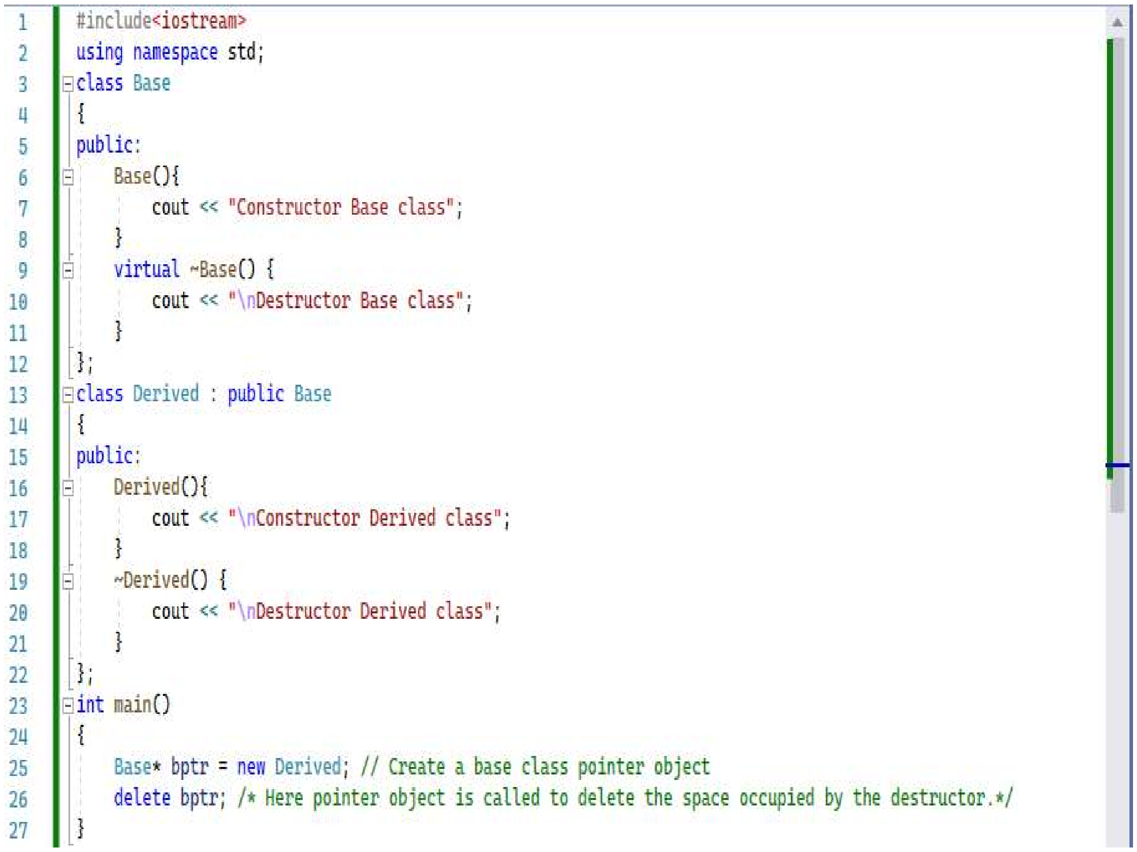
Output:



```
Constructor Base class
Constructor Derived class
Destructor Base class
```

Fig. 08 (without using Virtual Destructor)

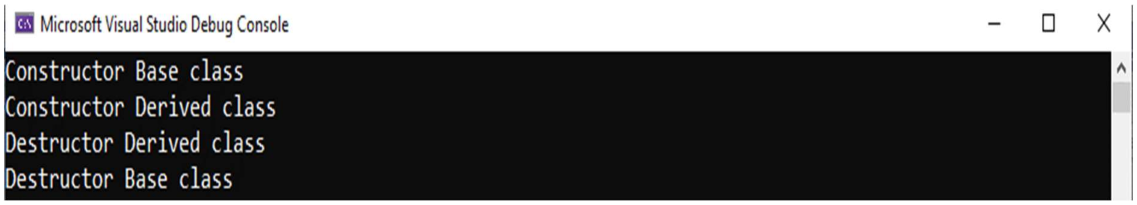
Example:



```cpp
#include<iostream>
using namespace std;
class Base
{
public:
    Base(){
        cout << "Constructor Base class";
    }
    virtual ~Base() {
        cout << "\nDestructor Base class";
    }
};
class Derived : public Base
{
public:
    Derived(){
        cout << "\nConstructor Derived class";
    }
    ~Derived() {
        cout << "\nDestructor Derived class";
    }
};
int main()
{
    Base* bptr = new Derived; // Create a base class pointer object
    delete bptr; /* Here pointer object is called to delete the space occupied by the destructor.*/
}
```

Fig. 09 (using Virtual Destructor)

Output:



```
Constructor Base class
Constructor Derived class
Destructor Derived class
Destructor Base class
```

Fig. 10 (using Virtual Destructor)

Abstract Class:

An abstract class has at least one pure virtual function. In other words, a function that has no definition. The abstract class's descendants must define the pure virtual function; otherwise, the subclass would become an abstract class. Following is the syntax for an abstract class:

class classname //abstract class{

//data members

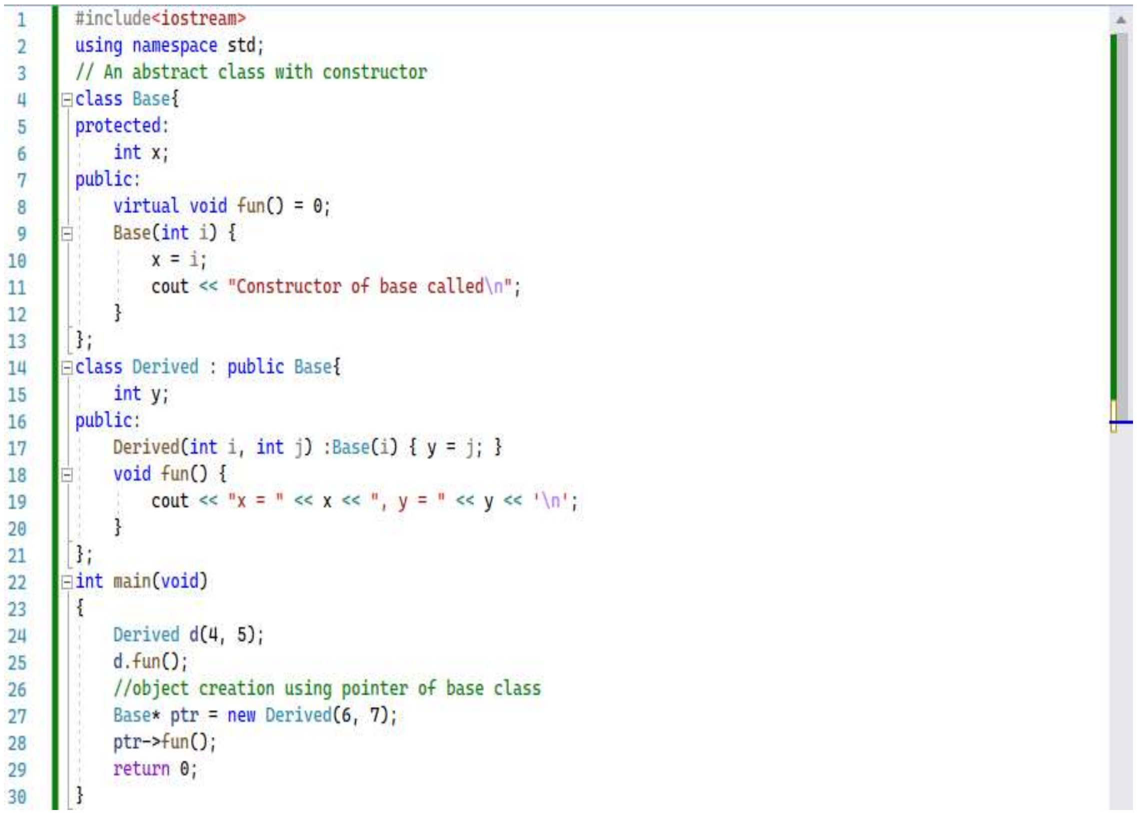
public:

//pure virtual function

/* Other members */

};

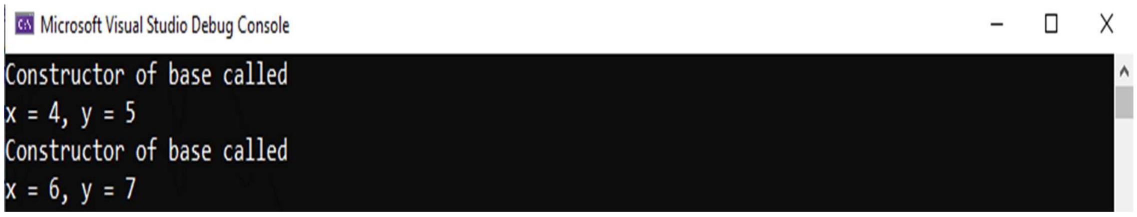
Example:



```
#include<iostream>
using namespace std;
// An abstract class with constructor
class Base{
protected:
    int x;
public:
    virtual void fun() = 0;
    Base(int i) {
        x = i;
        cout << "Constructor of base called\n";
    }
};
class Derived : public Base{
    int y;
public:
    Derived(int i, int j) :Base(i) { y = j; }
    void fun() {
        cout << "x = " << x << ", y = " << y << '\n';
    }
};
int main(void)
{
    Derived d(4, 5);
    d.fun();
    //object creation using pointer of base class
    Base* ptr = new Derived(6, 7);
    ptr->fun();
    return 0;
}
```

Fig. 11 (Abstract Class)

Output:



```
Constructor of base called
x = 4, y = 5
Constructor of base called
x = 6, y = 7
```

Fig. 12 (Abstract Class)

Task 01: Banking System **[Estimated time 40 minutes / 30 marks]**

Create a program that simulates a bank account system. The program should have the following features:

- The bank has different types of accounts: savings account and checking account.
- Each account has a unique account number, balance, and interest rate.
- The program should allow the user to create a new account, deposit or withdraw money from an existing account, calculate the interest earned on an account, and display the current balance and account information.
- The interest rate for each account type should be set by the bank and should be a virtual function in the account class, so that it can be overridden by each account type.
- The program should also have a function to display the list of accounts owned by a particular customer.

The program should have the following classes:

- **Account:** Represents a bank account with a unique account number, balance, and interest rate. Has functions to deposit or withdraw money, calculate the interest earned, and display the account information. Also has a virtual function to calculate the interest rate.
- **SavingsAccount:** Represents a savings account, which has a higher interest rate than a checking account. Overrides the virtual function to set the interest rate.
- **CheckingAccount:** Represents a checking account, which has a lower interest rate than a savings account. Overrides the virtual function to set the interest rate.
- **Customer:** Represents a customer with a unique ID and a list of accounts they own. Has functions to create accounts, deposit or withdraw money from accounts, and display the customer information. Also has a function to display the list of accounts owned by the customer.

Task 02: Calculator **[Estimated time 30 minutes / 20 marks]**

Create a C++ program that simulates a simple calculator. The program should have the following features:

- The calculator should be able to perform basic arithmetic operations: addition, subtraction, multiplication, and division.
- The program should allow the user to enter two numbers and choose an operation.
- The program should then display the result of the operation.
- Finally, the program should exit when the user chooses to quit.
- To demonstrate the use of virtual destructor, you can create a base class called **'Operation'**, which has virtual destructor. Then, create four derived classes which inherit from Operation and implement their respective operations. The four classes are:
    - Addition
    - Subtraction
    - Multiplication
    - Division
- When the user chooses an operation, the program should create an object of the corresponding derived class and call its calculate() method. Since Operation has a virtual destructor, the destructor of the derived class will be called automatically when the object goes out of scope.

Task 03: Student Record **[Estimated time 30 minutes / 20 marks]**

- By implementing pure virtual function, make a class **'Student'** with data members
  - name
  - department as protected type
- and two pure virtual member functions
  - get_data() "to input the student record"
  - show_data() "display student record".
- Derive three classes **'Medical'**, **'Engineering'**, and **'Science'** from Student class with its own member functions of get_data() "to input the student record" and show_data() "display student record" while using data members of student class.
- In main create an array of pointers of base class having size three and assign to each index remaining class objects addresses. By using for loop iterate array and take input then again display records using for loop.

Post-Lab Activities:

Task 01: Chess Game **[Estimated time 60 minutes / 40 marks]**

Create a C++ program that simulates a simple chess game. The program should have the following features:

- There are six types of pieces: pawn, rook, knight, bishop, queen, and king.
- Each piece has a unique ID, a position on the board, and a method to move to a new position.
- The program should allow the user to move a piece on the board, check if a move is valid, and determine if a piece is in check or checkmate.
- The program should also allow the user to add a new piece, remove an existing piece, and display the current state of the chess board.
- To demonstrate the use of abstract class, you can create an abstract base class called **'ChessPiece'**, which has a pure virtual function called **'isValidMove()'**.
- Create six derived classes: **'Pawn'**, **'Rook'**, **'Knight'**, **'Bishop'**, **'Queen'**, and **'King'**, which inherit from **'ChessPiece'** and implement their respective **'isValidMove()'** methods.
- When the user moves a piece on the board, the program should call the isValidMove() method of the corresponding derived class to check if the move is valid.
- Additionally, the program should have a function to determine if a piece is in check. Since the rules for check are the same for all types of pieces, you can define these functions in the **'ChessPiece'** base class and override them in the derived classes if necessary.

Submissions:

- For In-Lab Activity:
  - Save the files on your PC.
  - TA's will evaluate the tasks offline.
- For Pre-Lab & Post-Lab Activity:
  - Submit the .cpp file on Google Classroom and name it to your roll no.

Evaluations Metric:

- All the lab tasks will be evaluated offline by TA's

Division of Pre-Lab marks:	**[20 marks]**
Task 01: Calculate Area	[10 marks]
Task 02: Animal Sound	[10 marks]
Division of In-Lab marks:	**[70 marks]**
Task 01: Banking System	[30 marks]
Task 02: Calculator	[20 marks]
Task 03: Student Record	[20 marks]
Division of Post-Lab marks:	**[40 marks]**
Task 01: Chess Game	[40 marks]

References and Additional Material:

- **Function Overriding**
  https://www.programiz.com/cpp-programming/function-overriding
- **Virtual Functions**
  https://www.programiz.com/cpp-programming/virtual-functions
- **Virtual Destructor**
  https://www.geeksforgeeks.org/virtual-destructor/
- **Abstract Class and Pure Virtual Functions**
  https://www.programiz.com/cpp-programming/pure-virtual-funtion

Lab Time Activity Simulation Log:

• Slot – 01 – 00:00 – 00:15:	Class Settlement
• Slot – 02 – 00:15 – 00:40:	In-Lab Task
• Slot – 03 – 00:40 – 01:20:	In-Lab Task
• Slot – 04 – 01:20 – 02:20:	In-Lab Task
• Slot – 05 – 02:20 – 02:45:	Evaluation of Lab Tasks
• Slot – 06 – 02:45 – 03:00:	Discussion on Post-Lab Task