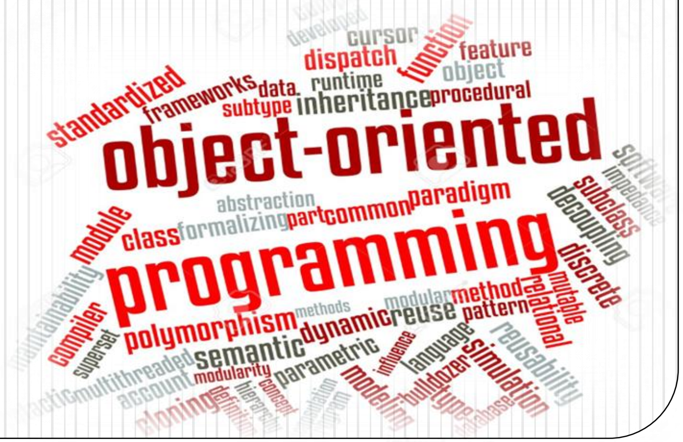
F.Y.B.Sc.IT – SEM II

OBJECT ORIENTED PROGRAMMING WITH C++ (PUSIT206T)

BY,
PROF. NIKITA MADWAL

UNIT 3

9. WORKING WITH INHERITANCE IN C++
10. POINTERS TO OBJECTS AND VIRTUAL FUNCTIONS
11. INPUT-OUTPUT AND MANIPULATORS IN C++



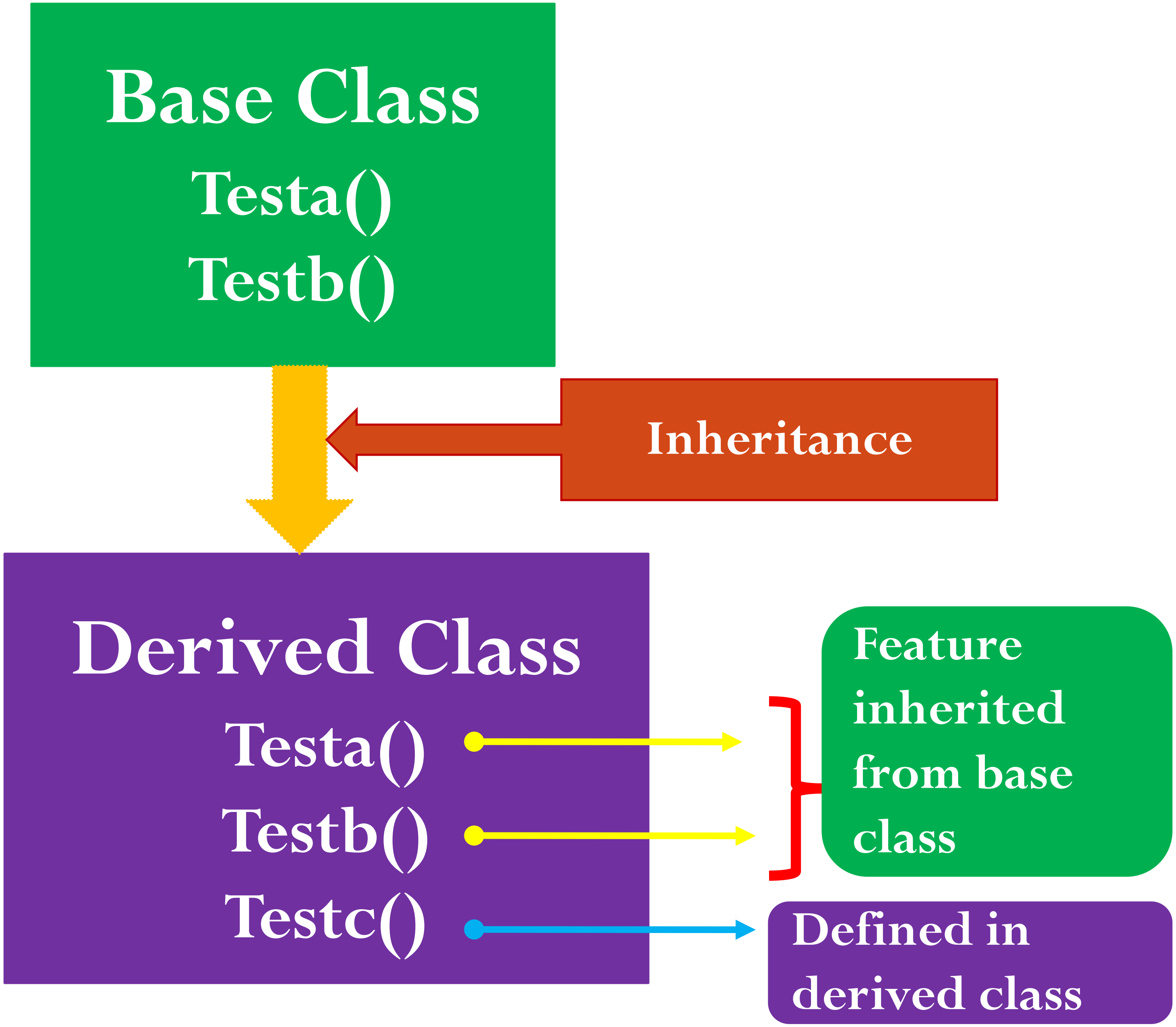
9.
WORKING WITH INHERITANCE IN C++

Introduction

- The mechanism of **deriving a new class** from an **old one** is called **inheritance** (or derivation)
- The **old class** is referred to as the **base class** or **super class** or **parents class** and the **new one** is called the **derived class** or **subclass** or **child class**
- **Derived class :** class which inherits the members of another class
- **Base class :** the class whose members are inherited
- **Existing classes (old class)** are the **main components** of **inheritance**

- New classes are created from existing one (old class)
- **Properties** of existing classes are simply **extended** to the new classes
- The **derived class** **inherits** **some or all** of the **traits** (property or characteristics) from the **base class**
- The real life example of inheritance is child and parents, all or most of the properties of parents are inherited by their child

Fig.



Base Class

Testa()

Testb()

Inheritance

Derived Class

Testa()

Testb()

Testc()

Feature inherited from base class

Defined in derived class

Advantages

- **Reusability :** One of the key benefits of inheritance is to minimize the amount of duplicate code in an application by sharing common code amongst several subclasses (**When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused** (i.e. **we need not write the same code again in child class**))
- **Save time and effort :** The concept of reusability achieved by inheritance saves the programmer time and effort
- **Maintainability:** It is easy to debug a program when divided in parts. Inheritance provides an opportunity to capture the program.(i.e. **as existing code is reused ,it leads to less development and maintenance costs**)

Syntax of Inheritance

```
class base_class
{
    .................. //Body of  base class
};
class derived_class : access_specifier base_class
{
   ...................//Body of  derived class
};
```

Defining Derived Classes

- A derived class is specified by **defining its relationship with the base class in addition to its own details**.
- The general syntax of defining a derived class is as follows:

```
class derived_classname  :  access_specifier baseclass_name
{
      --------  // body of derived class
};
```

- The colon (:) indicates that the a class name is derived from the base class name.
- The **access specifiers** or the **visibility mode** is **optional** and, if present, may be **public**, **private** or **protected**. By **default** it is **private**.
- While **defining** a **subclass** like this, the **super class** must be **already defined** or **at least declared <u>before</u>** the **subclass declaration**

- Visibility mode describes the status of derived features.
- e.g.

```
class xyz                    //base class
{
          private :
          public :
          protected :
         members of xyz
};
class abc : public xyz     //public derivation
{
      members of abc
};
class abc : xyz     //private derivation (by default)
{
      members of abc
};
class abc : protected xyz     //protected derivation
{
    members of abc
};
```

Access Specifiers and Inheritance

- Access mode is used to specify, the mode in which the **properties** of **superclass (base class)** will be inherited into **subclass (derived class)**, public, private or protected.
- **Public** members visible to all classes
- **Private** members are visible only to the class to which they belong
- **Protected** members are visible only to the class to which they belong and derived classes

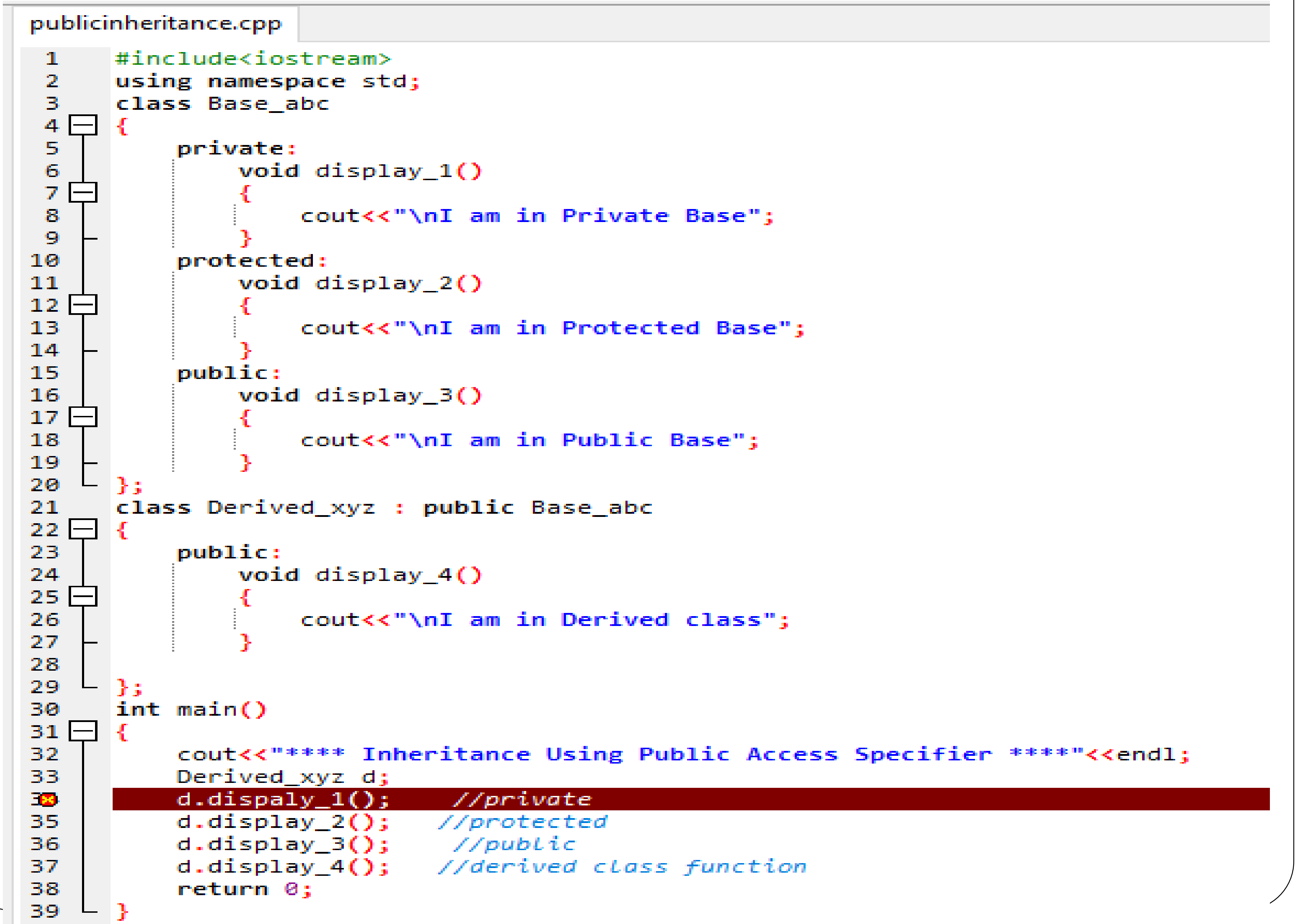
NOTE : All members of a class **except** **Private**, are inherited

Public Inheritance

- This is the most used inheritance mode.
- When the base class is ***publicly*** *inherited* **by** the derived class ,
  - All the **public members** of base class becomes **public members** of derived class.
  - All the **protected members** of base class becomes **protected members** of derived class.
  - **Private** members are **never inherited**.
- When the base class is ***publicly*** *inherited* , the derived class **object can access** only the **public members** of the base class. The **protected members of** the base class are **inaccessible by** the **objects** of derived class
- Syntax :

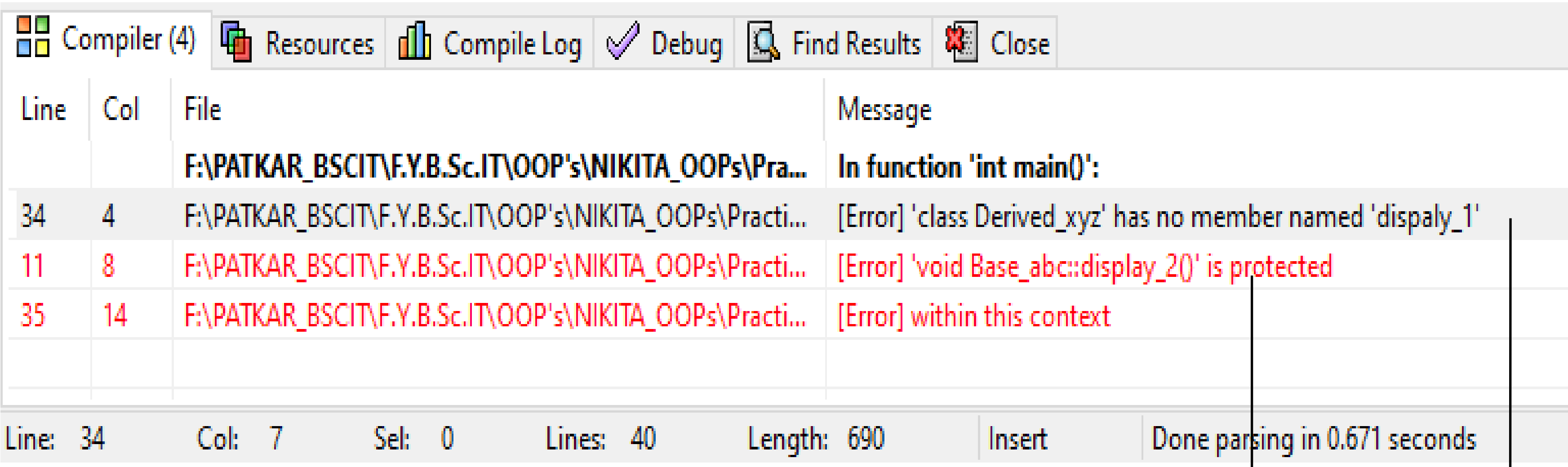
```
class derived-class: public base-class
```

Invoking all the access specifiers function (Pubic, Private and Protected)



```cpp
#include<iostream>
using namespace std;
class Base_abc
{
    private:
        void display_1()
        {
            cout<<"\nI am in Private Base";
        }
    protected:
        void display_2()
        {
            cout<<"\nI am in Protected Base";
        }
    public:
        void display_3()
        {
            cout<<"\nI am in Public Base";
        }
};
class Derived_xyz : public Base_abc
{
    public:
        void display_4()
        {
            cout<<"\nI am in Derived class";
        }

};
int main()
{
    cout<<"**** Inheritance Using Public Access Specifier ****"<<endl;
    Derived_xyz d;
    d.dispaly_1();    //private
    d.display_2();   //protected
    d.display_3();    //public
    d.display_4();   //derived class function
    return 0;
}
```

Error :

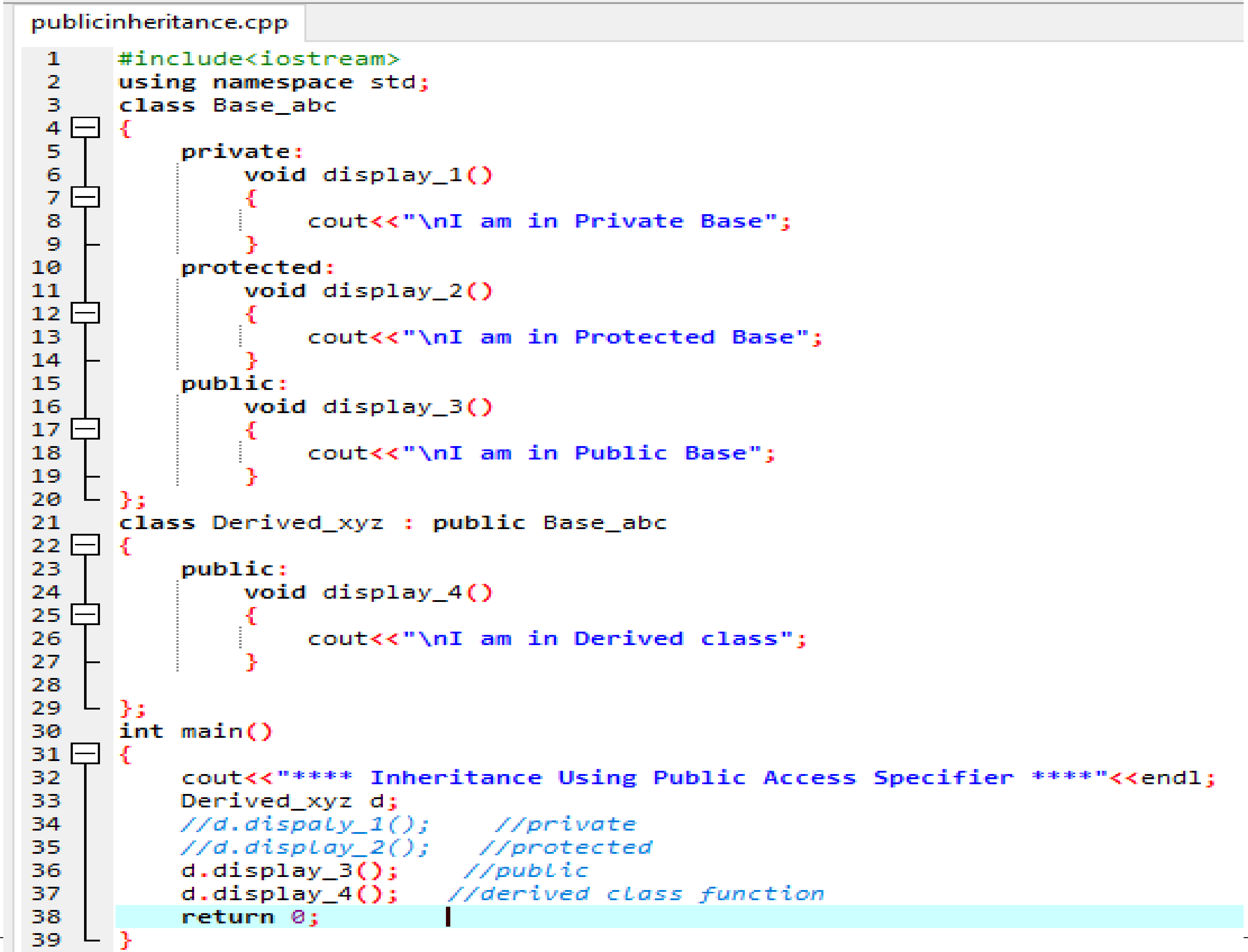


```
Line  Col  File                                             Message
           F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Pra...  In function 'int main()':
34    4    F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Error] 'class Derived_xyz' has no member named 'dispaly_1'
11    8    F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Error] 'void Base_abc::display_2()' is protected
35    14   F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Error] within this context

Line: 34    Col: 7    Sel: 0    Lines: 40    Length: 690    Insert    Done parsing in 0.671 seconds
```

Private function display_1() is not accessible

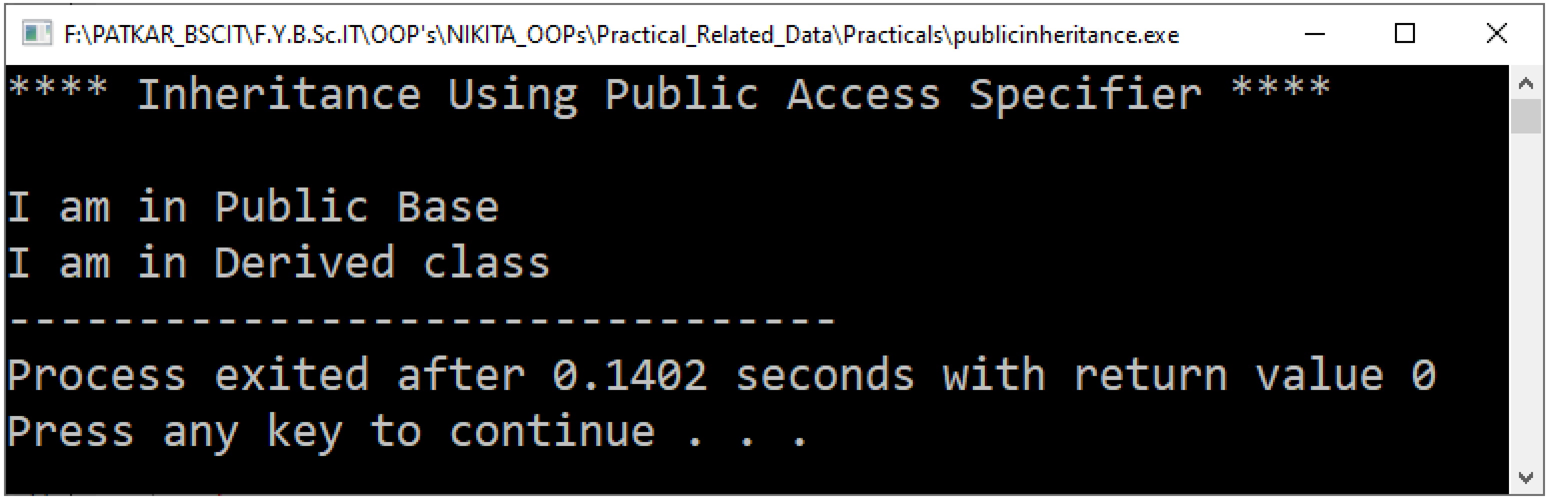
Protected function display_2() is not accessible

After disabling display_1()[Private] and dispalay_2() [Protected] function



```cpp
#include<iostream>
using namespace std;
class Base_abc
{
	private:
		void display_1()
		{
			cout<<"\nI am in Private Base";
		}
	protected:
		void display_2()
		{
			cout<<"\nI am in Protected Base";
		}
	public:
		void display_3()
		{
			cout<<"\nI am in Public Base";
		}
};
class Derived_xyz : public Base_abc
{
	public:
		void display_4()
		{
			cout<<"\nI am in Derived class";
		}

};
int main()
{
	cout<<"**** Inheritance Using Public Access Specifier ****"<<endl;
	Derived_xyz d;
	//d.dispaly_1();    //private
	//d.display_2();   //protected
	d.display_3();    //public
	d.display_4();   //derived class function
	return 0;
}
```

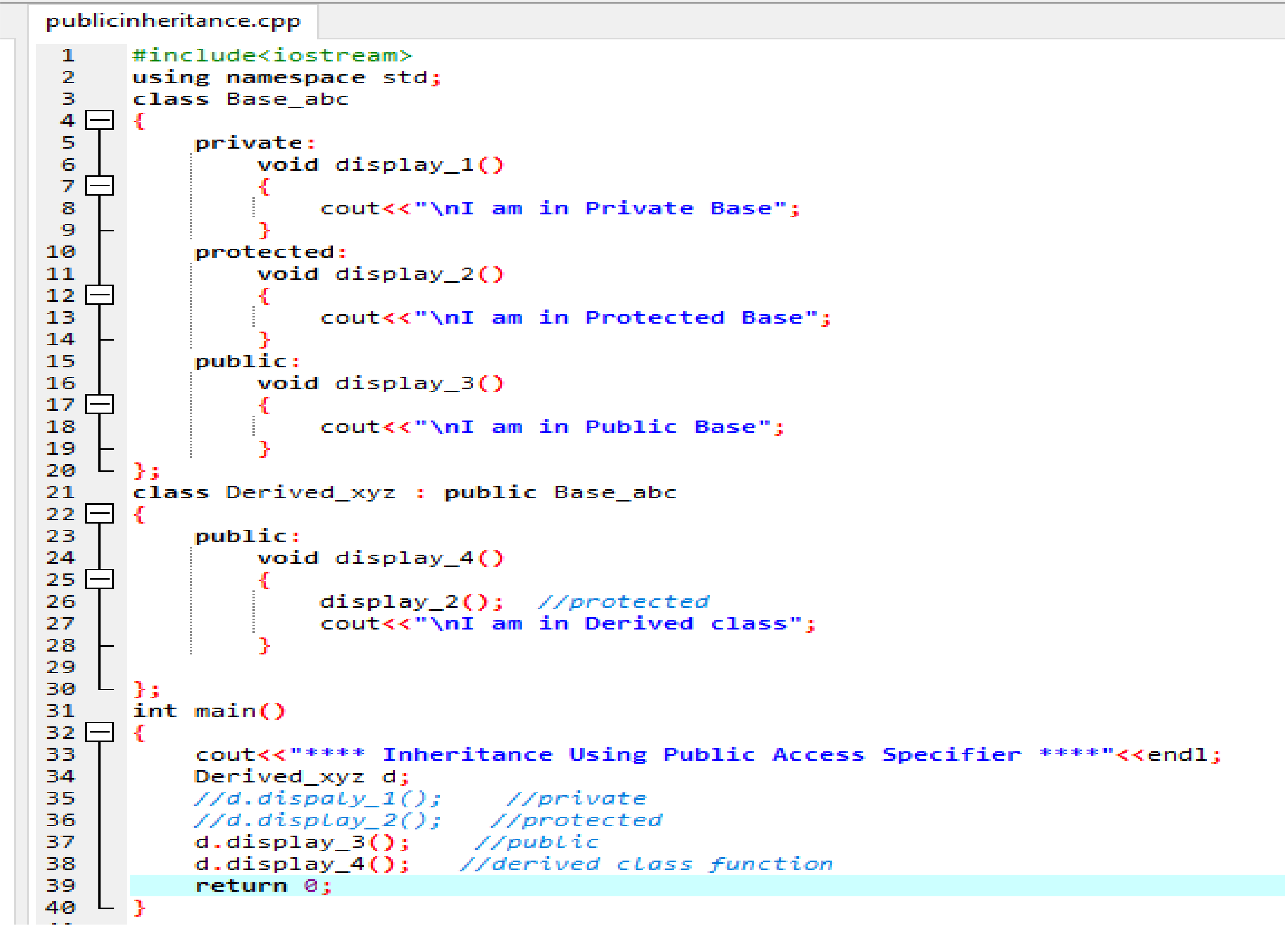
Output :



F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\publicinheritance.exe

```
**** Inheritance Using Public Access Specifier ****

I am in Public Base
I am in Derived class
--------------------------------
Process exited after 0.1402 seconds with return value 0
Press any key to continue . . .
```

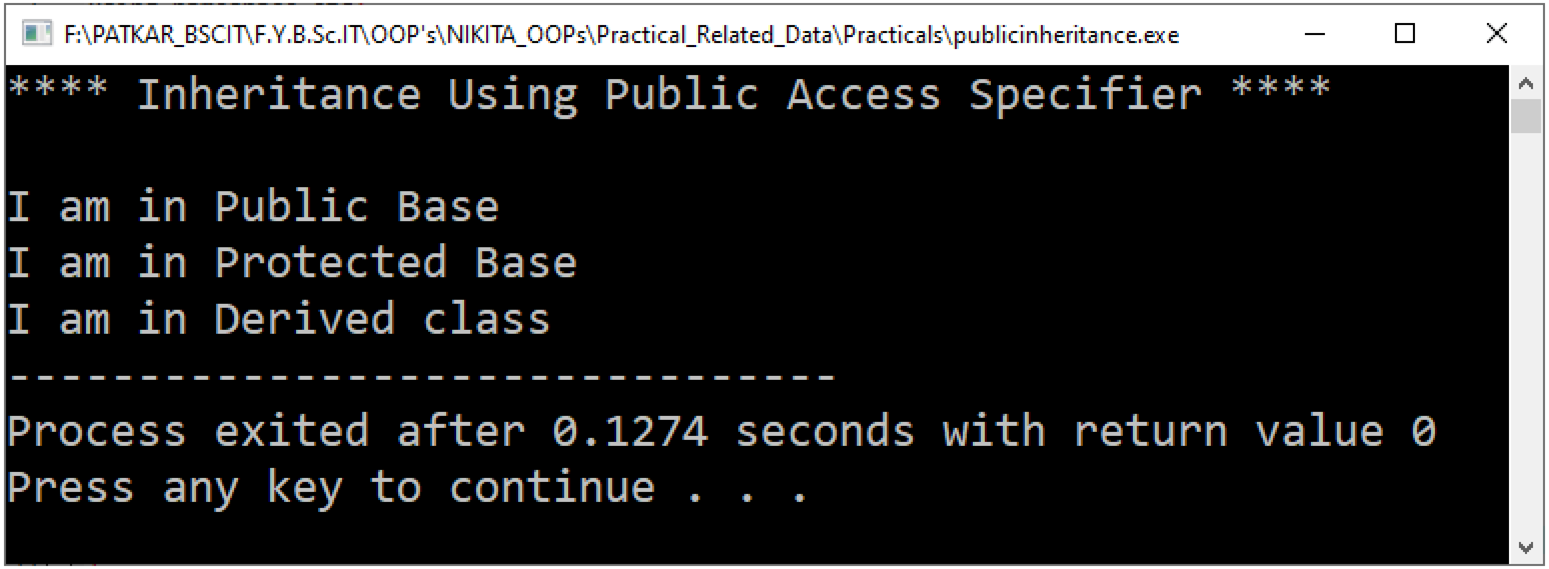
Solution :



publicinheritance.cpp

```cpp
#include<iostream>
using namespace std;
class Base_abc
{
    private:
        void display_1()
        {
            cout<<"\nI am in Private Base";
        }
    protected:
        void display_2()
        {
            cout<<"\nI am in Protected Base";
        }
    public:
        void display_3()
        {
            cout<<"\nI am in Public Base";
        }
};
class Derived_xyz : public Base_abc
{
    public:
        void display_4()
        {
            display_2();  //protected
            cout<<"\nI am in Derived class";
        }

};
int main()
{
    cout<<"**** Inheritance Using Public Access Specifier ****"<<endl;
    Derived_xyz d;
    //d.dispaly_1();     //private
    //d.display_2();    //protected
    d.display_3();    //public
    d.display_4();   //derived class function
    return 0;
}
```

Output :


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\publicinheritance.exe

```
**** Inheritance Using Public Access Specifier ****

I am in Public Base
I am in Protected Base
I am in Derived class
--------------------------------
Process exited after 0.1274 seconds with return value 0
Press any key to continue . . .
```

If you want to display protected function data using *Public inheritance* then it can be called into derived class Public function

Private Inheritance

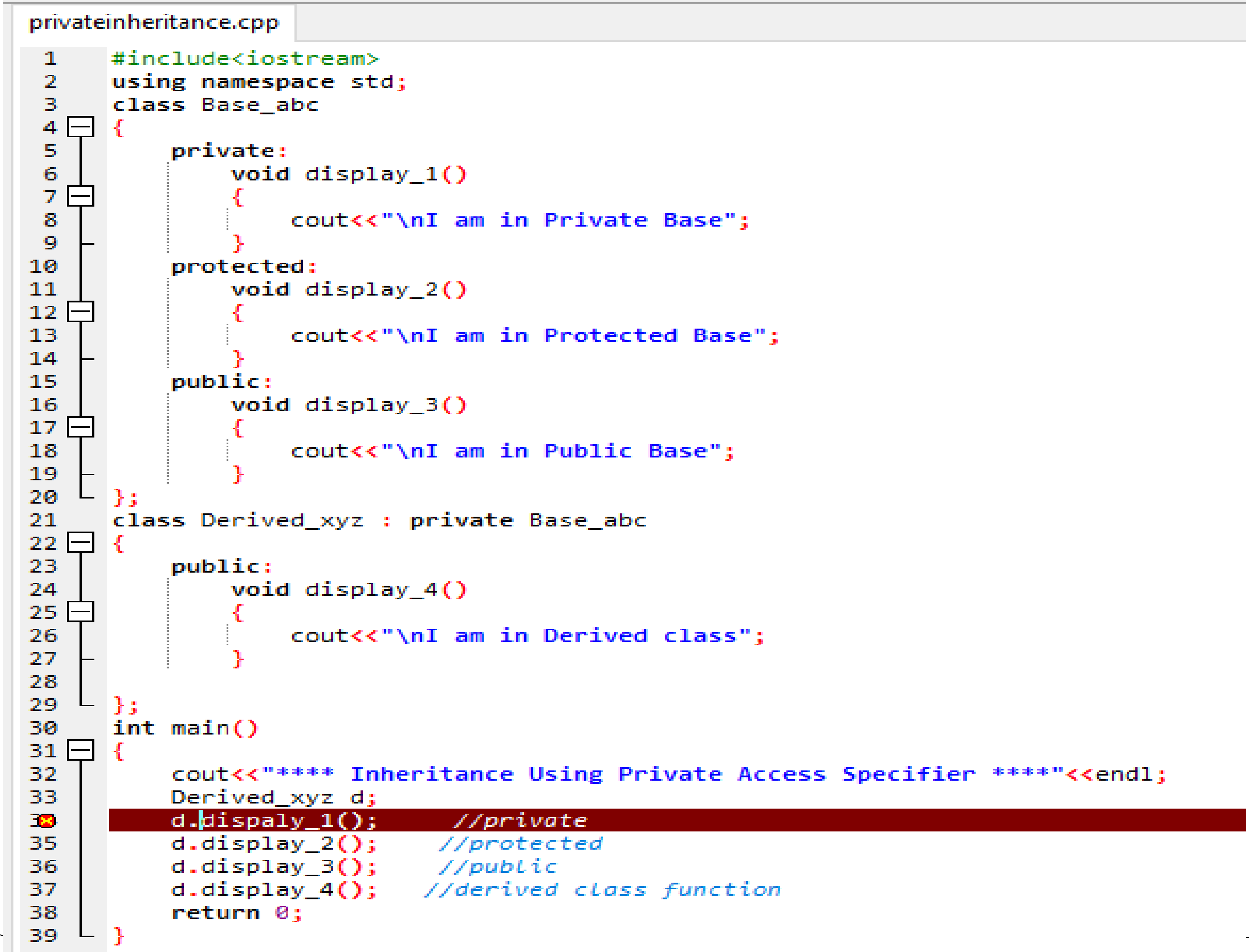
- All members of a class **<u>except</u>** **Private**, are inherited
- When the base class is ***privately*** *inherited* **by** the derived class,
  - **both <u>public members</u>** and **<u>protected members</u>** of the base class **becomes** the **<u>private</u>** members of the derived class.
  - **Private** members are **never inherited.**
- When the base class is ***privately*** *inherited* , the derived class **<u>object</u> <u>cannot access</u>** the **public** and **protected** members of the base class.
- Syntax:

```
class derived-class: private base-class
```

OR

```
class derived-class: base-class
```

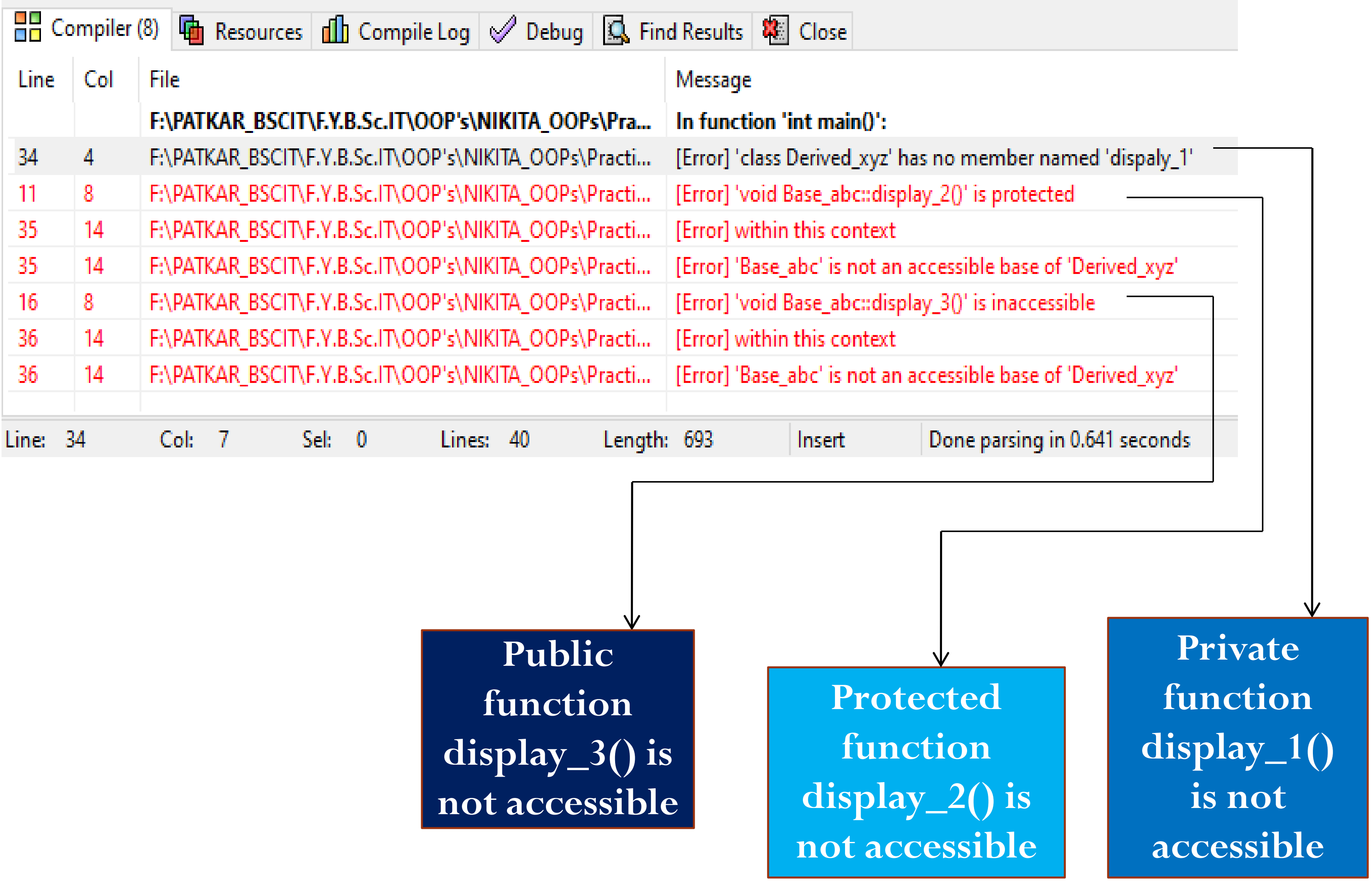
Invoking all the access specifiers function (Pubic, Private and Protected)



privateinheritance.cpp

```cpp
#include<iostream>
using namespace std;
class Base_abc
{
	private:
		void display_1()
		{
			cout<<"\nI am in Private Base";
		}
	protected:
		void display_2()
		{
			cout<<"\nI am in Protected Base";
		}
	public:
		void display_3()
		{
			cout<<"\nI am in Public Base";
		}
};
class Derived_xyz : private Base_abc
{
	public:
		void display_4()
		{
			cout<<"\nI am in Derived class";
		}

};
int main()
{
	cout<<"**** Inheritance Using Private Access Specifier ****"<<endl;
	Derived_xyz d;
	d.dispaly_1();       //private
	d.display_2();     //protected
	d.display_3();     //public
	d.display_4();   //derived class function
	return 0;
}
```

Error :



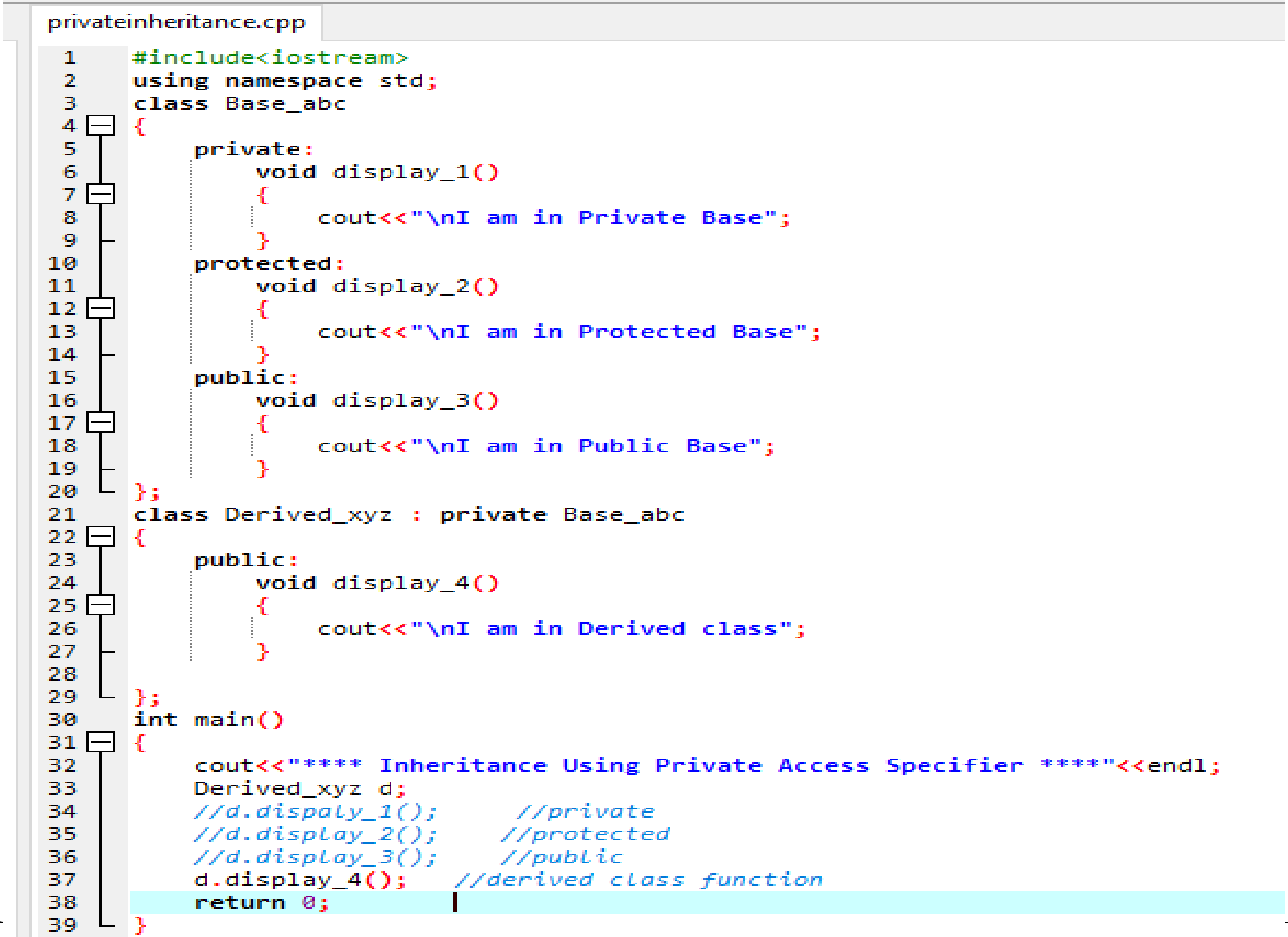
```
Line  Col  File                                              Message
           F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Pra...  In function 'int main()':
34    4    F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Error] 'class Derived_xyz' has no member named 'dispaly_1'
11    8    F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Error] 'void Base_abc::display_2()' is protected
35    14   F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Error] within this context
35    14   F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Error] 'Base_abc' is not an accessible base of 'Derived_xyz'
16    8    F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Error] 'void Base_abc::display_3()' is inaccessible
36    14   F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Error] within this context
36    14   F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Error] 'Base_abc' is not an accessible base of 'Derived_xyz'

Line:  34      Col:  7      Sel:  0      Lines:  40      Length:  693      Insert      Done parsing in 0.641 seconds
```

Public function display_3() is not accessible

Protected function display_2() is not accessible

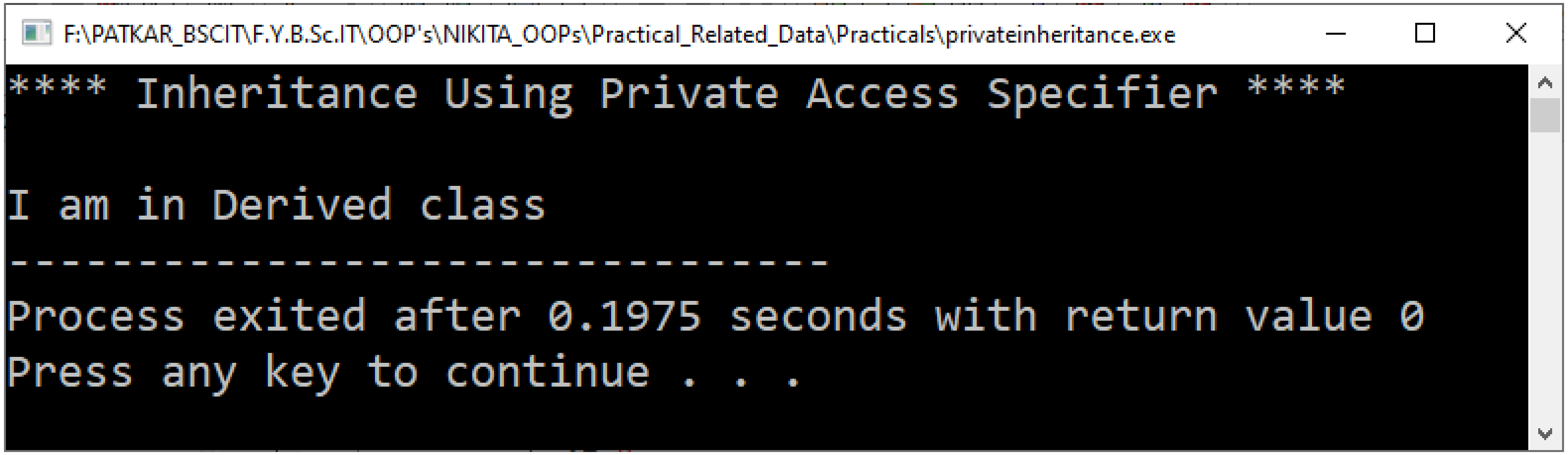
Private function display_1() is not accessible

After disabling display_1()[Private] ,dispalay_2() [Protected] , display_3()[Public] f unction



```cpp
#include<iostream>
using namespace std;
class Base_abc
{
    private:
        void display_1()
        {
            cout<<"\nI am in Private Base";
        }
    protected:
        void display_2()
        {
            cout<<"\nI am in Protected Base";
        }
    public:
        void display_3()
        {
            cout<<"\nI am in Public Base";
        }
};
class Derived_xyz : private Base_abc
{
    public:
        void display_4()
        {
            cout<<"\nI am in Derived class";
        }

};
int main()
{
    cout<<"***** Inheritance Using Private Access Specifier ****"<<endl;
    Derived_xyz d;
    //d.dispaly_1();     //private
    //d.display_2();    //protected
    //d.display_3();    //public
    d.display_4();   //derived class function
    return 0;
}
```

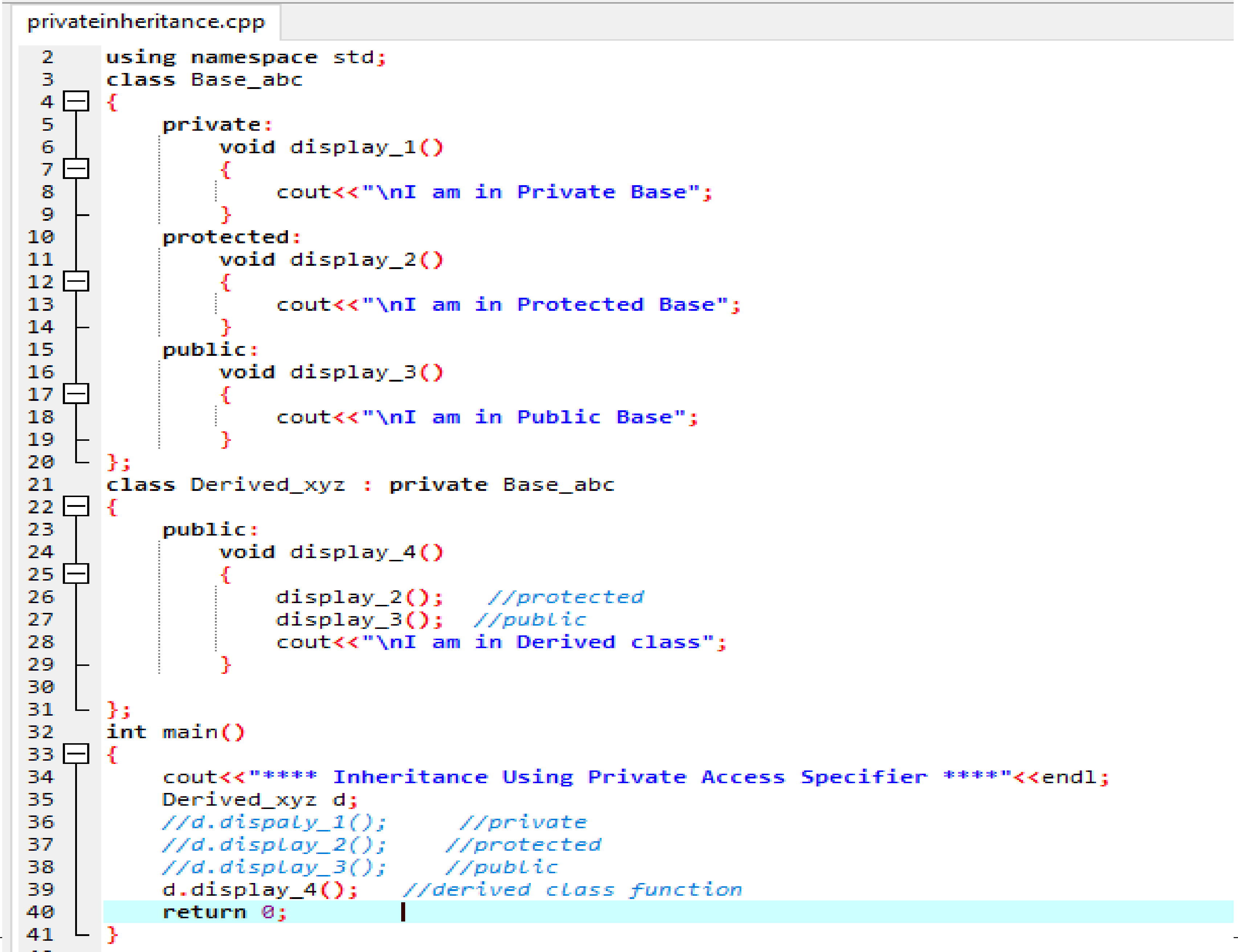
Output :



F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\privateinheritance.exe

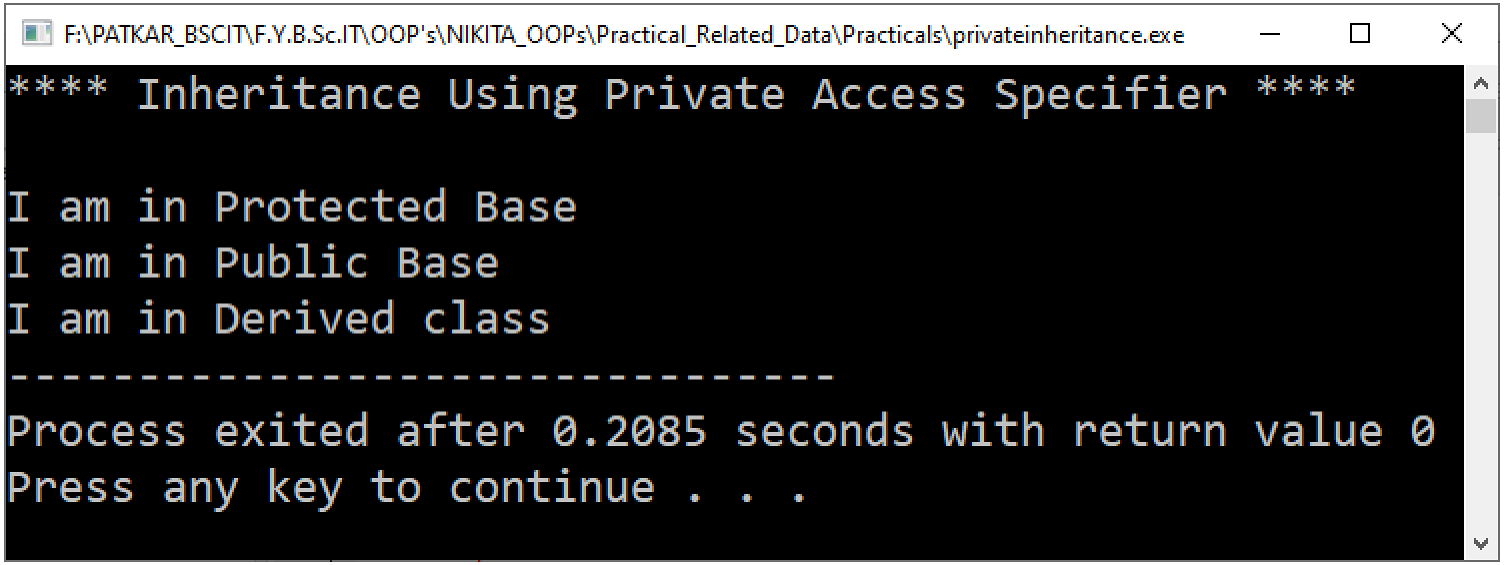
```
**** Inheritance Using Private Access Specifier ****

I am in Derived class
--------------------------------
Process exited after 0.1975 seconds with return value 0
Press any key to continue . . .
```

Solution :



```cpp
using namespace std;
class Base_abc
{
    private:
        void display_1()
        {
            cout<<"\nI am in Private Base";
        }
    protected:
        void display_2()
        {
            cout<<"\nI am in Protected Base";
        }
    public:
        void display_3()
        {
            cout<<"\nI am in Public Base";
        }
};
class Derived_xyz : private Base_abc
{
    public:
        void display_4()
        {
            display_2();   //protected
            display_3();  //public
            cout<<"\nI am in Derived class";
        }

};
int main()
{
    cout<<"**** Inheritance Using Private Access Specifier ****"<<endl;
    Derived_xyz d;
    //d.dispaly_1();     //private
    //d.display_2();    //protected
    //d.display_3();    //public
    d.display_4();   //derived class function
    return 0;
}
```

Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\privateinheritance.exe

```
**** Inheritance Using Private Access Specifier ****

I am in Protected Base
I am in Public Base
I am in Derived class
--------------------------------
Process exited after 0.2085 seconds with return value 0
Press any key to continue . . .
```

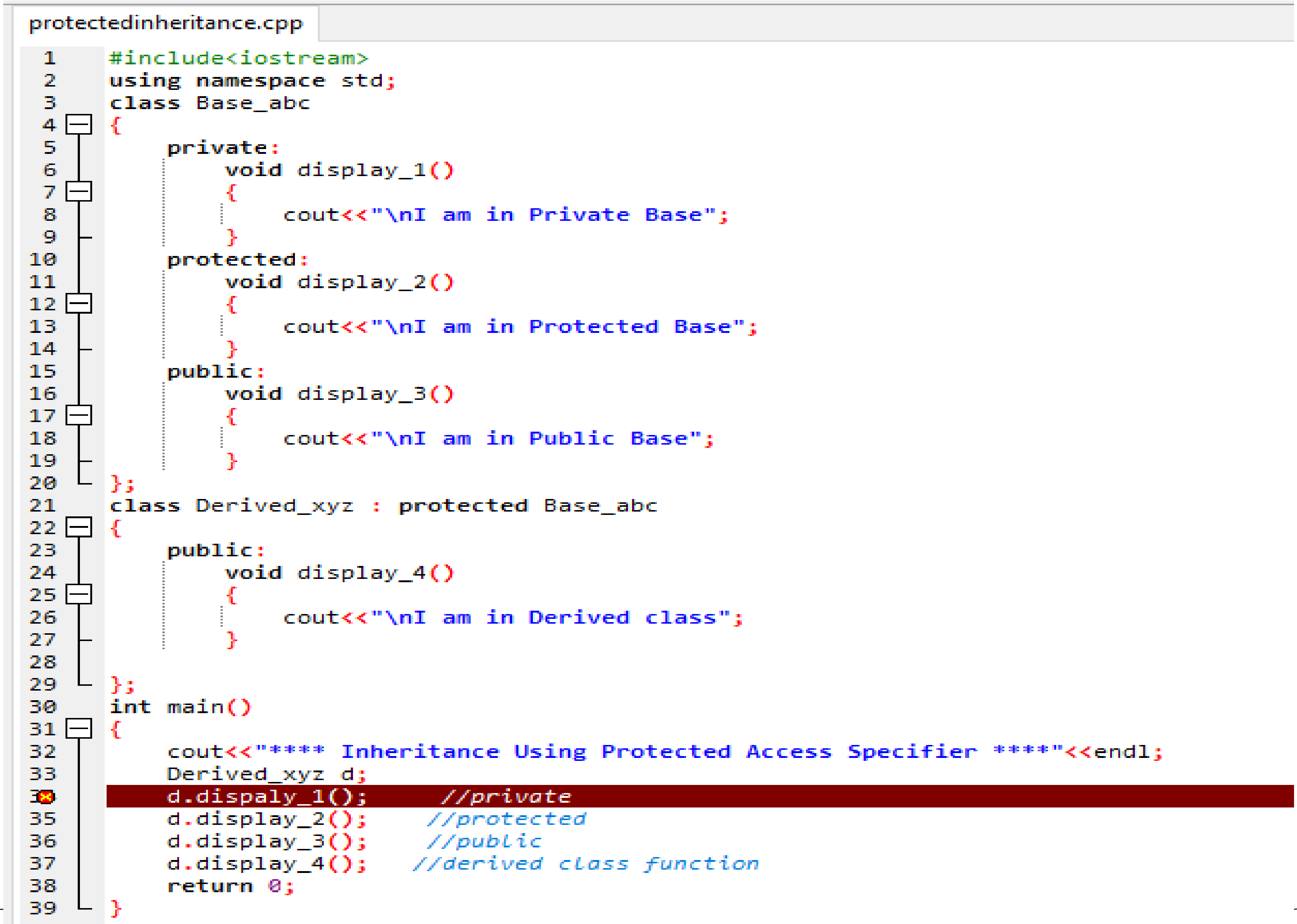
If you want to display public function's and protected function's data using *Private inheritance* then it can be called into derived class Public function

Protected Inheritance

- When the base class is ***protectedly*** inherited by the derived class,
- both **public members** and **protected members** **of** the base class becomes the **protected** members **of the** derived class.
- **Private** members are **never inherited.**
- When the base class is ***protectedly*** inherited, the **objects** of the derived class **cannot access** the **public** and **protected** members of the base class.
- Syntax:

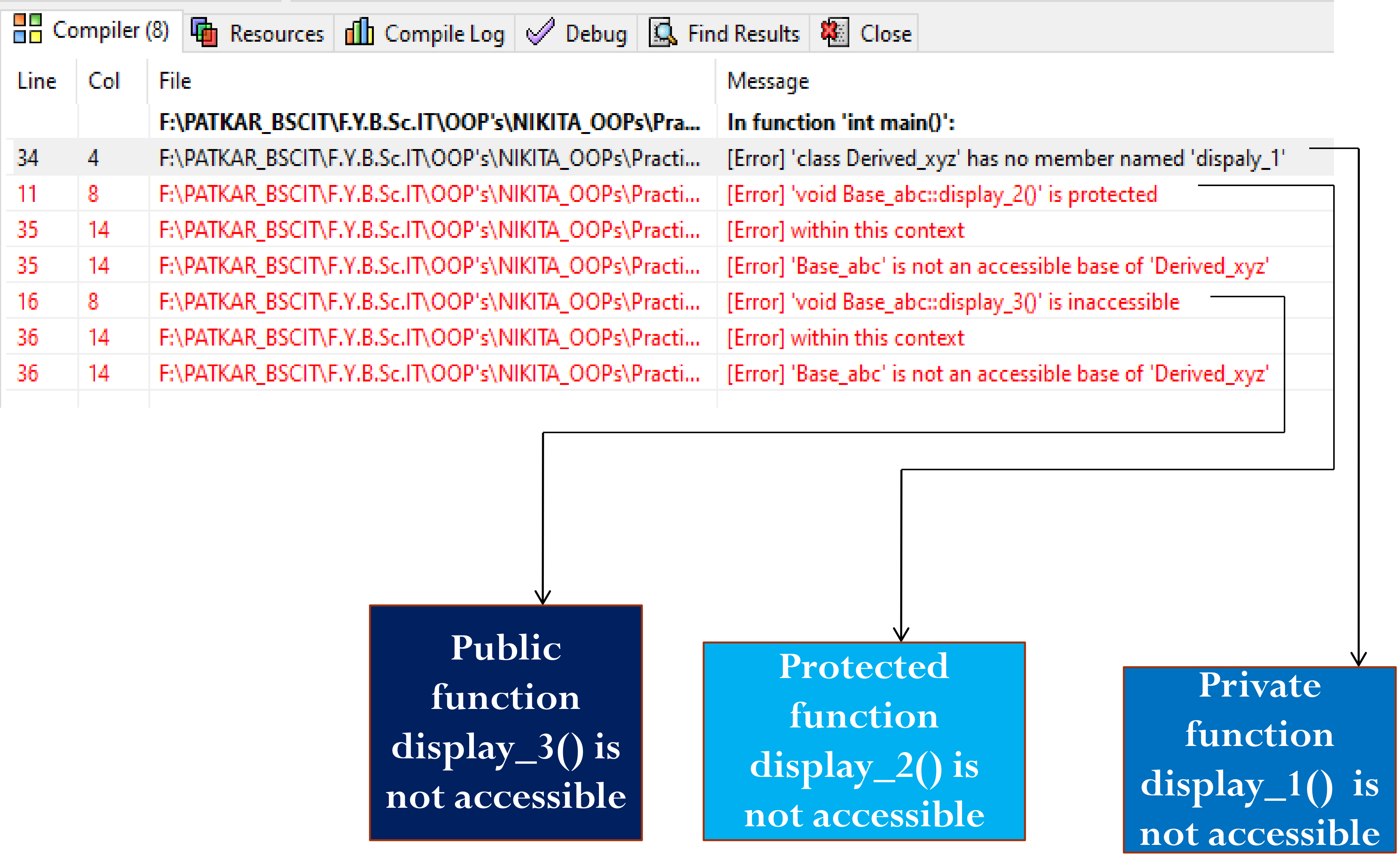
```
class derived-class: protected base-class
```

Invoking all the access specifiers function (Pubic, Private and Protected)



```cpp
#include<iostream>
using namespace std;
class Base_abc
{
	private:
		void display_1()
		{
			cout<<"\nI am in Private Base";
		}
	protected:
		void display_2()
		{
			cout<<"\nI am in Protected Base";
		}
	public:
		void display_3()
		{
			cout<<"\nI am in Public Base";
		}
};
class Derived_xyz : protected Base_abc
{
	public:
		void display_4()
		{
			cout<<"\nI am in Derived class";
		}

};
int main()
{
	cout<<"**** Inheritance Using Protected Access Specifier ****"<<endl;
	Derived_xyz d;
	d.dispaly_1();		//private
	d.display_2();		//protected
	d.display_3();		//public
	d.display_4();	//derived class function
	return 0;
}
```

Error :



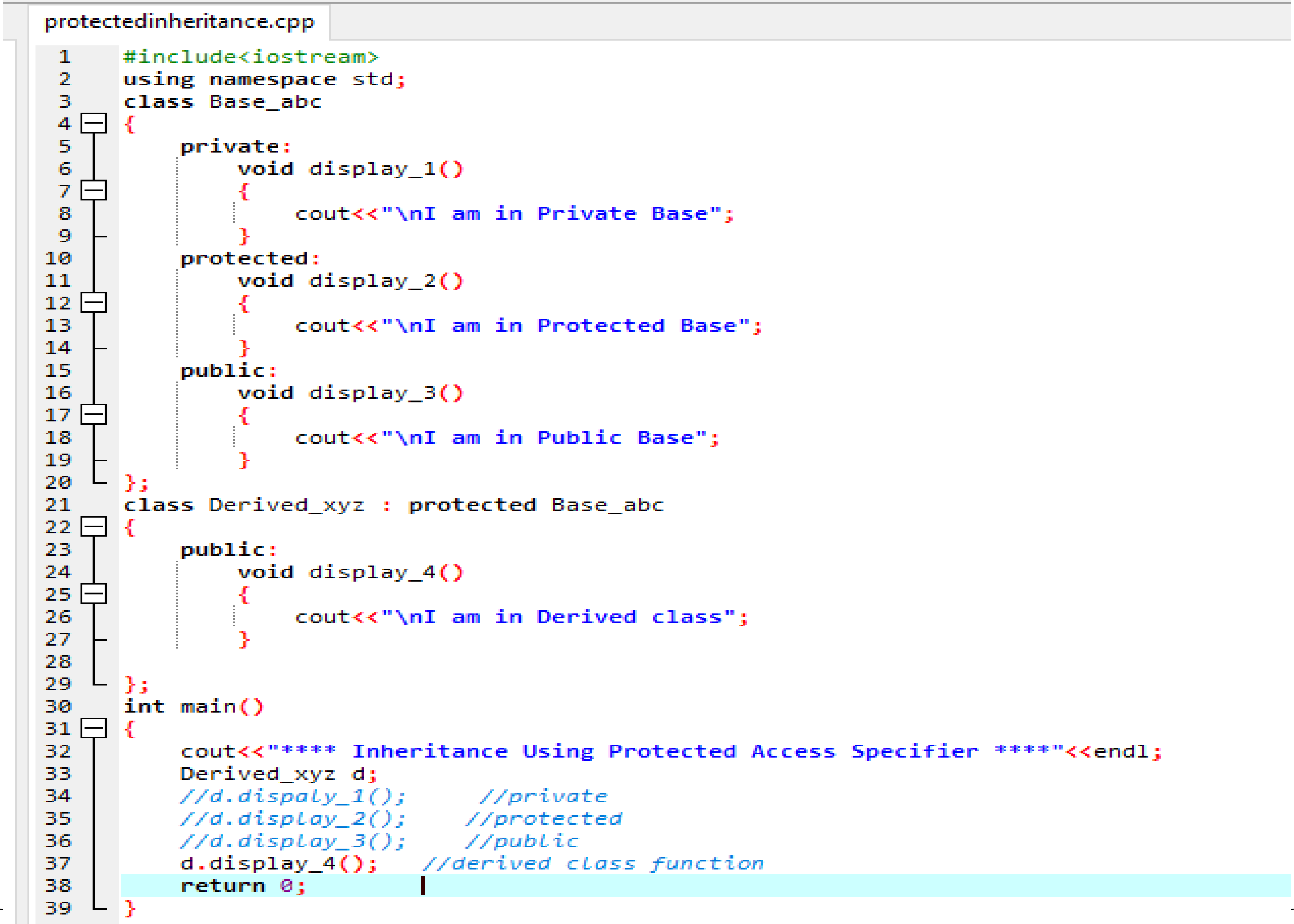
```
Line  Col  File                                              Message
           F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Pra...  In function 'int main()':
34    4    F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Error] 'class Derived_xyz' has no member named 'dispaly_1'
11    8    F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Error] 'void Base_abc::display_2()' is protected
35    14   F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Error] within this context
35    14   F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Error] 'Base_abc' is not an accessible base of 'Derived_xyz'
16    8    F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Error] 'void Base_abc::display_3()' is inaccessible
36    14   F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Error] within this context
36    14   F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Error] 'Base_abc' is not an accessible base of 'Derived_xyz'
```

Public function display_3() is not accessible

Protected function display_2() is not accessible

Private function display_1() is not accessible

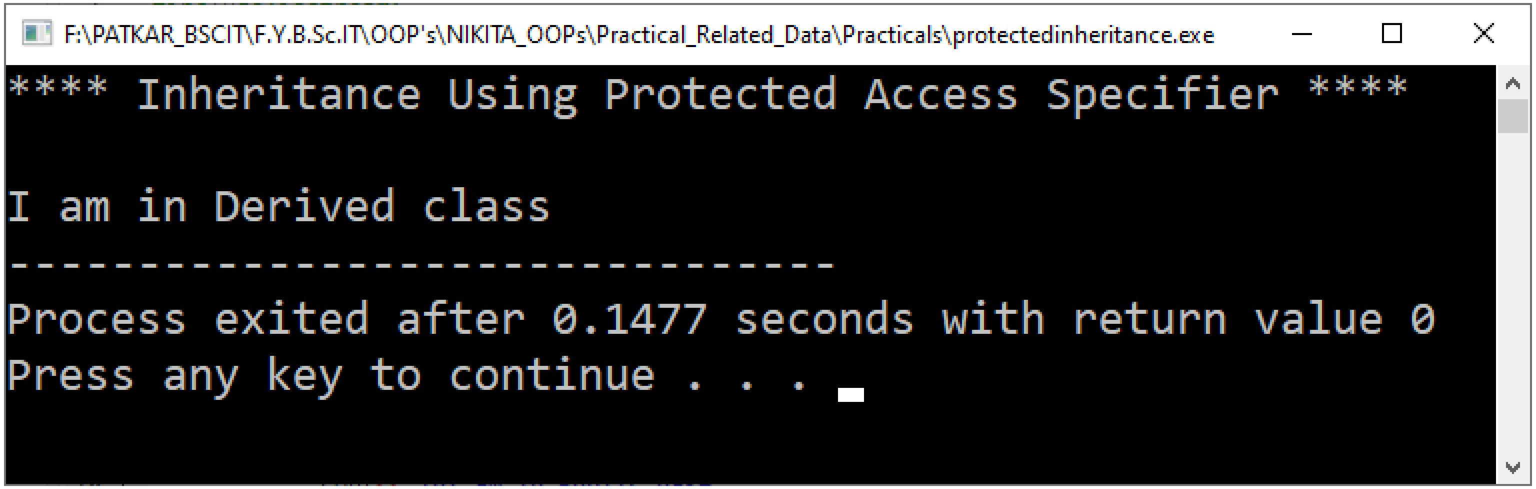
After disabling display_1()[Private] ,dispalay_2() [Protected] , display_3()[Public] f unction



protectedinheritance.cpp

```cpp
#include<iostream>
using namespace std;
class Base_abc
{
    private:
        void display_1()
        {
            cout<<"\nI am in Private Base";
        }
    protected:
        void display_2()
        {
            cout<<"\nI am in Protected Base";
        }
    public:
        void display_3()
        {
            cout<<"\nI am in Public Base";
        }
};
class Derived_xyz : protected Base_abc
{
    public:
        void display_4()
        {
            cout<<"\nI am in Derived class";
        }

};
int main()
{
    cout<<"**** Inheritance Using Protected Access Specifier ****"<<endl;
    Derived_xyz d;
    //d.dispaly_1();      //private
    //d.display_2();     //protected
    //d.display_3();     //public
    d.display_4();   //derived class function
    return 0;
}
```

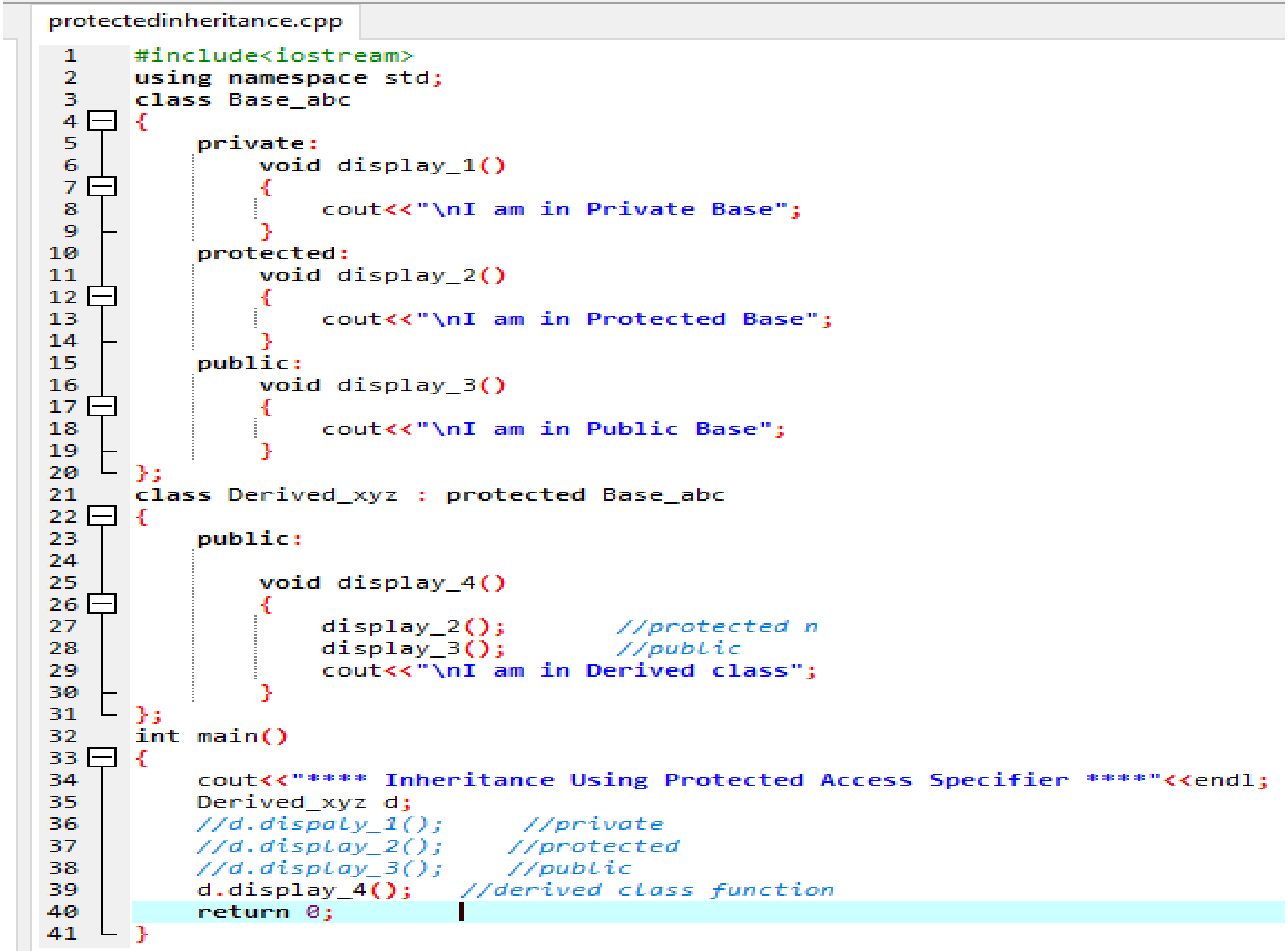
Output :



F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\protectedinheritance.exe

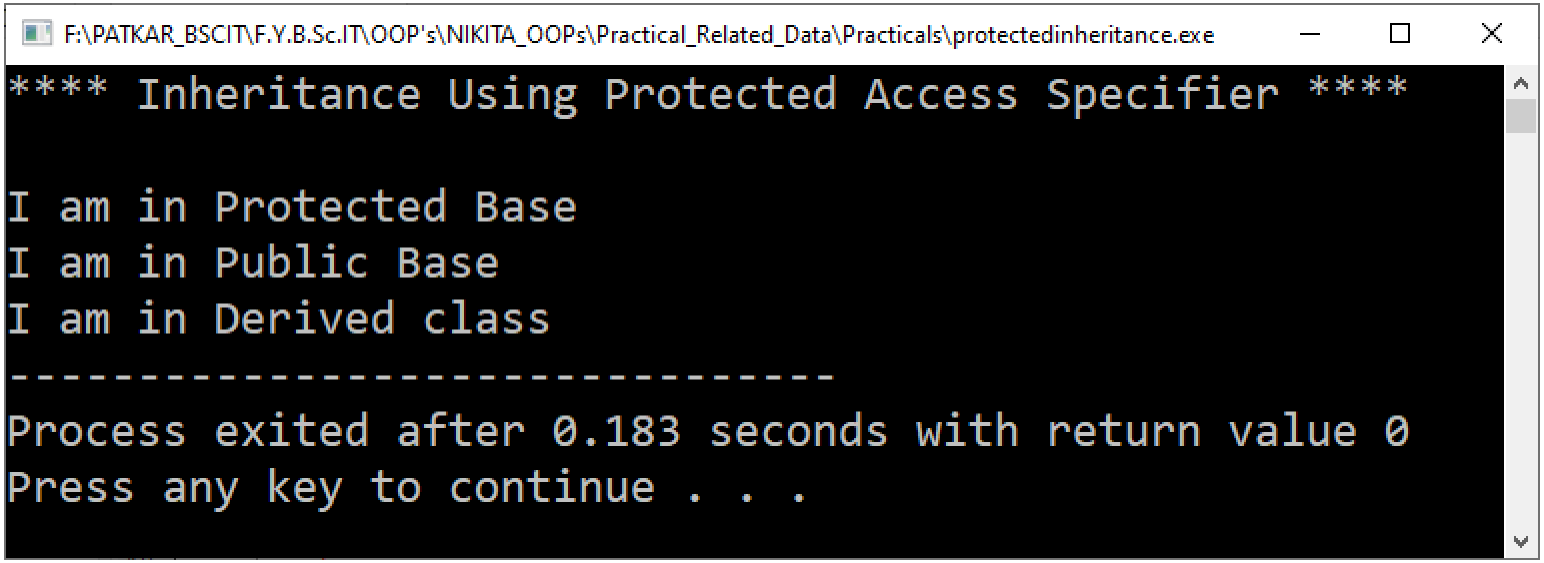
```
**** Inheritance Using Protected Access Specifier ****

I am in Derived class
--------------------------------
Process exited after 0.1477 seconds with return value 0
Press any key to continue . . .
```

Solution :



```
#include<iostream>
using namespace std;
class Base_abc
{
    private:
        void display_1()
        {
            cout<<"\nI am in Private Base";
        }
    protected:
        void display_2()
        {
            cout<<"\nI am in Protected Base";
        }
    public:
        void display_3()
        {
            cout<<"\nI am in Public Base";
        }
};
class Derived_xyz : protected Base_abc
{
    public:

        void display_4()
        {
            display_2();        //protected n
            display_3();        //public
            cout<<"\nI am in Derived class";
        }
};
int main()
{
    cout<<"**** Inheritance Using Protected Access Specifier ****"<<endl;
    Derived_xyz d;
    //d.dispaly_1();      //private
    //d.display_2();     //protected
    //d.display_3();     //public
    d.display_4();   //derived class function
    return 0;
}
```

Output :



F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\protectedinheritance.exe

```
**** Inheritance Using Protected Access Specifier ****

I am in Protected Base
I am in Public Base
I am in Derived class
--------------------------------
Process exited after 0.183 seconds with return value 0
Press any key to continue . . .
```

If you want to display public function's and protected function's data using *Protected inheritance* then it can be called into derived class Public function

Table showing all the visibility mode

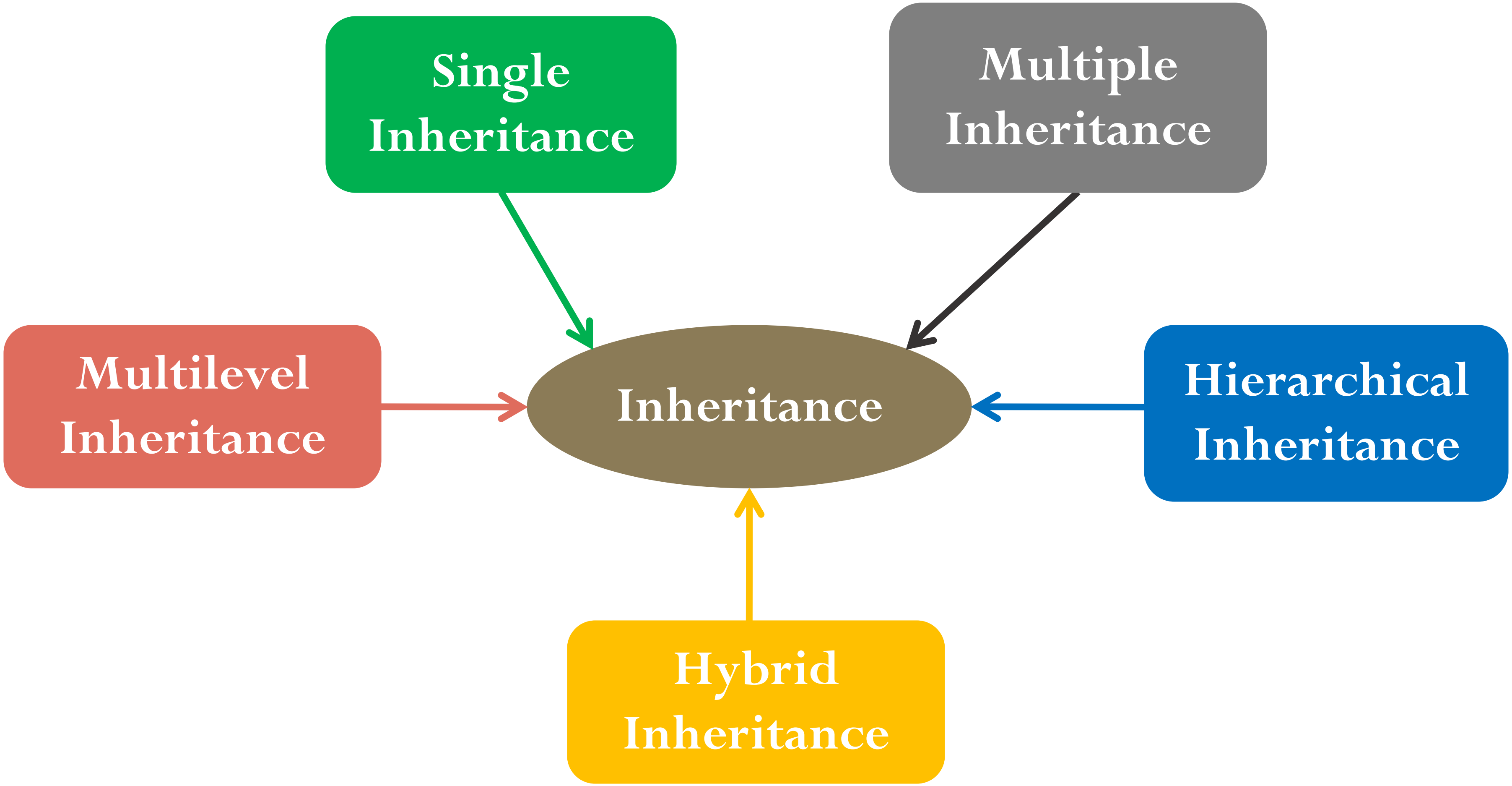
	Derived Class Access Mode		
Base Class Access Mode (Base Class Members)	**Public Derivation (Inheritance)**	**Private Derivation (Inheritance)**	**Protected Derivation (Inheritance)**
Private	Not Inheritable (Private)	Not Inheritable (Private)	Not Inheritable (Private)
Public	Public	Private	Protected
Protected	Protected	Private	Protected

- **Public** Inheritance
  - If a derived class is inherited **from** a ***public*** base class, the base class's **public members become public** and **protected members** become **protected in** the derived class

- **Private** Inheritance
  - If a derived class is inherited **from** a ***private*** base class, **both public** and **protected members of** the base class **become private** in the derived class

- **Protected** Inheritance
  - If derived class is inherited **from** a ***protected*** base class, the base class's **public** as well as **protected members become protected** in the derived class

Types of Inheritance



Single Inheritance

Multiple Inheritance

Multilevel Inheritance

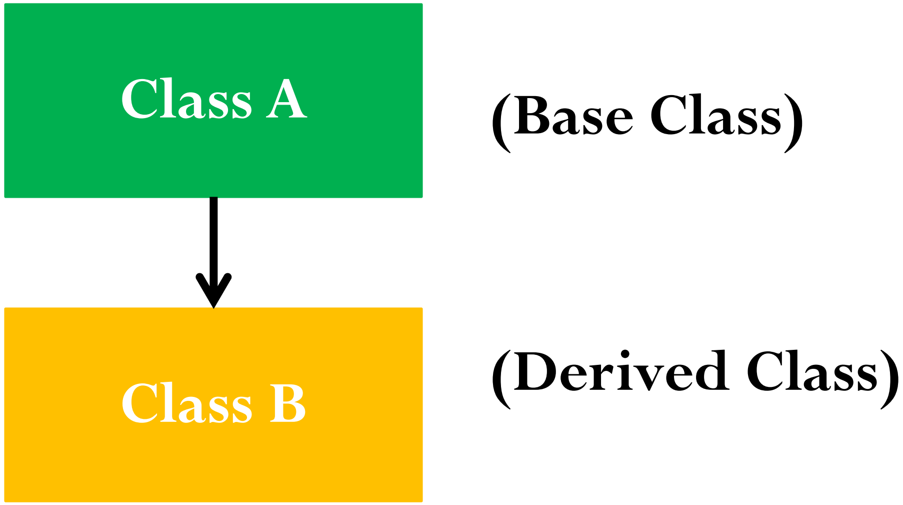
Inheritance

Hierarchical Inheritance

Hybrid Inheritance

Single Inheritance

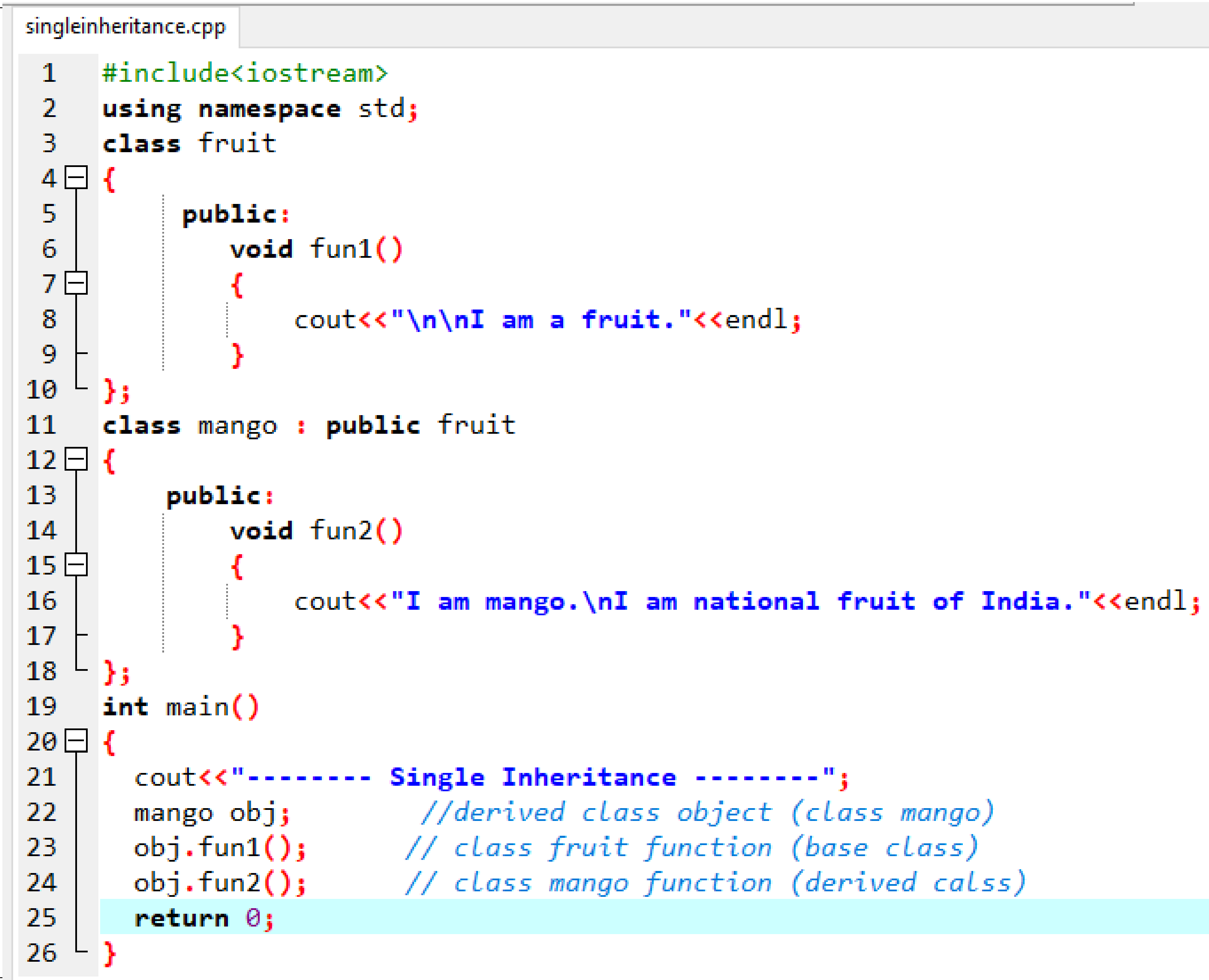
- In this type of inheritance **one** derived class inherits from **only one** base class. It is the most simplest form of Inheritance.
- A **derived class** with only **one base class** is called as **Single Inheritance**



- Syntax :

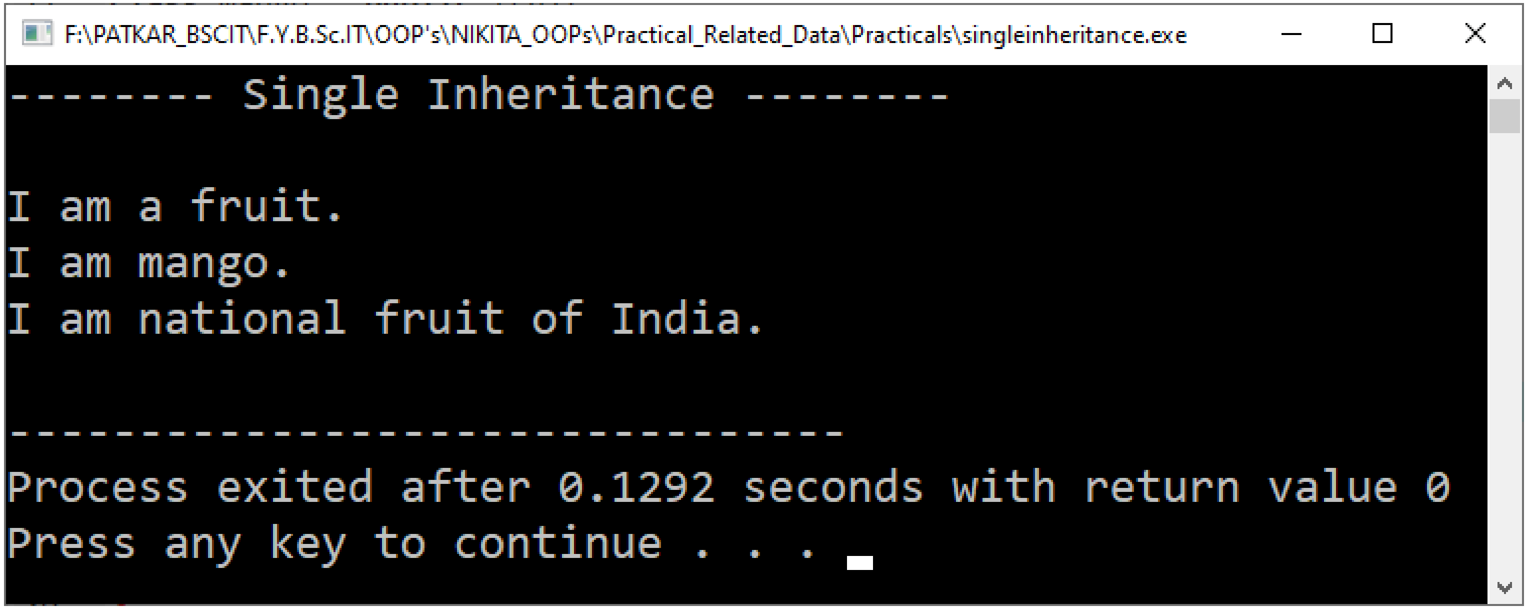
```
class base
{
            //Body of the base class
};
class derived : access_specifier base
{
            //Body of the derived class
};
```

e.g.



```cpp
#include<iostream>
using namespace std;
class fruit
{
     public:
        void fun1()
        {
            cout<<"\n\nI am a fruit."<<endl;
        }
};
class mango : public fruit
{
    public:
        void fun2()
        {
            cout<<"I am mango.\nI am national fruit of India."<<endl;
        }
};
int main()
{
  cout<<"-------- Single Inheritance --------";
  mango obj;        //derived class object (class mango)
  obj.fun1();      // class fruit function (base class)
  obj.fun2();      // class mango function (derived calss)
  return 0;
}
```

Output :

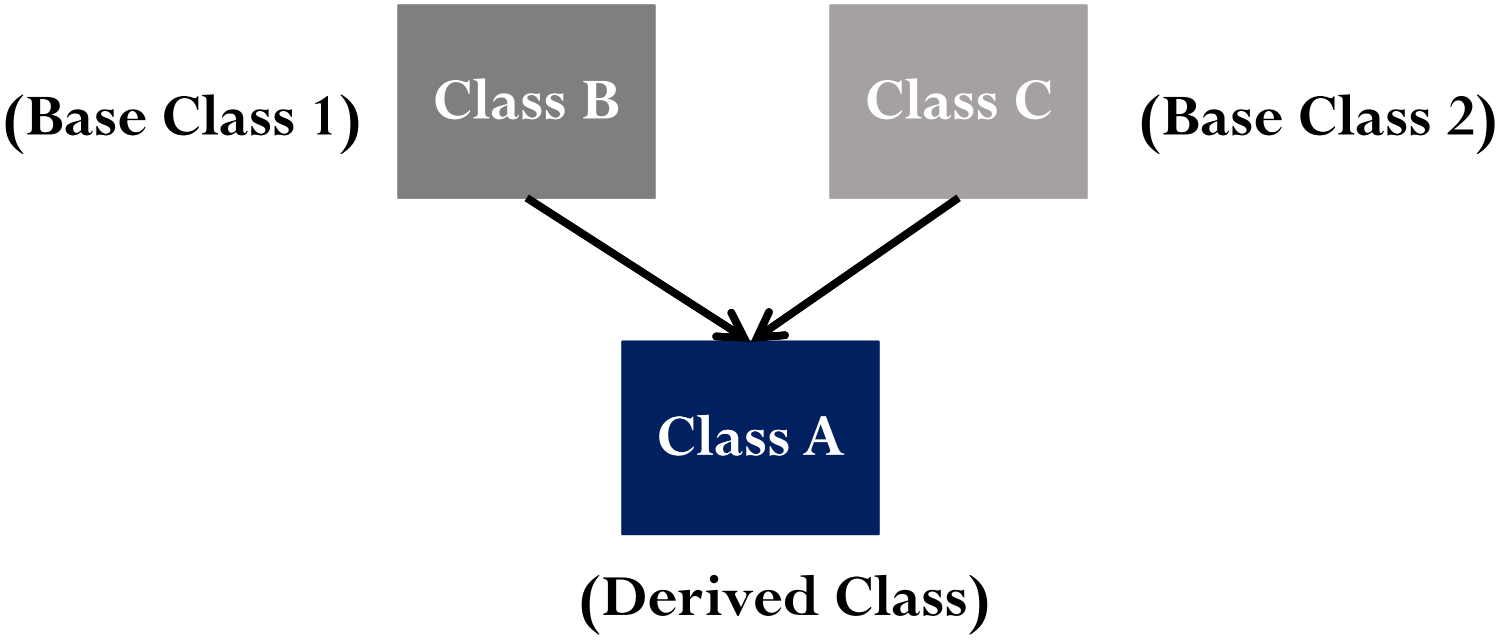


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\singleinheritance.exe

```
-------- Single Inheritance --------

I am a fruit.
I am mango.
I am national fruit of India.


--------------------------------
Process exited after 0.1292 seconds with return value 0
Press any key to continue . . .
```

Multiple Inheritance

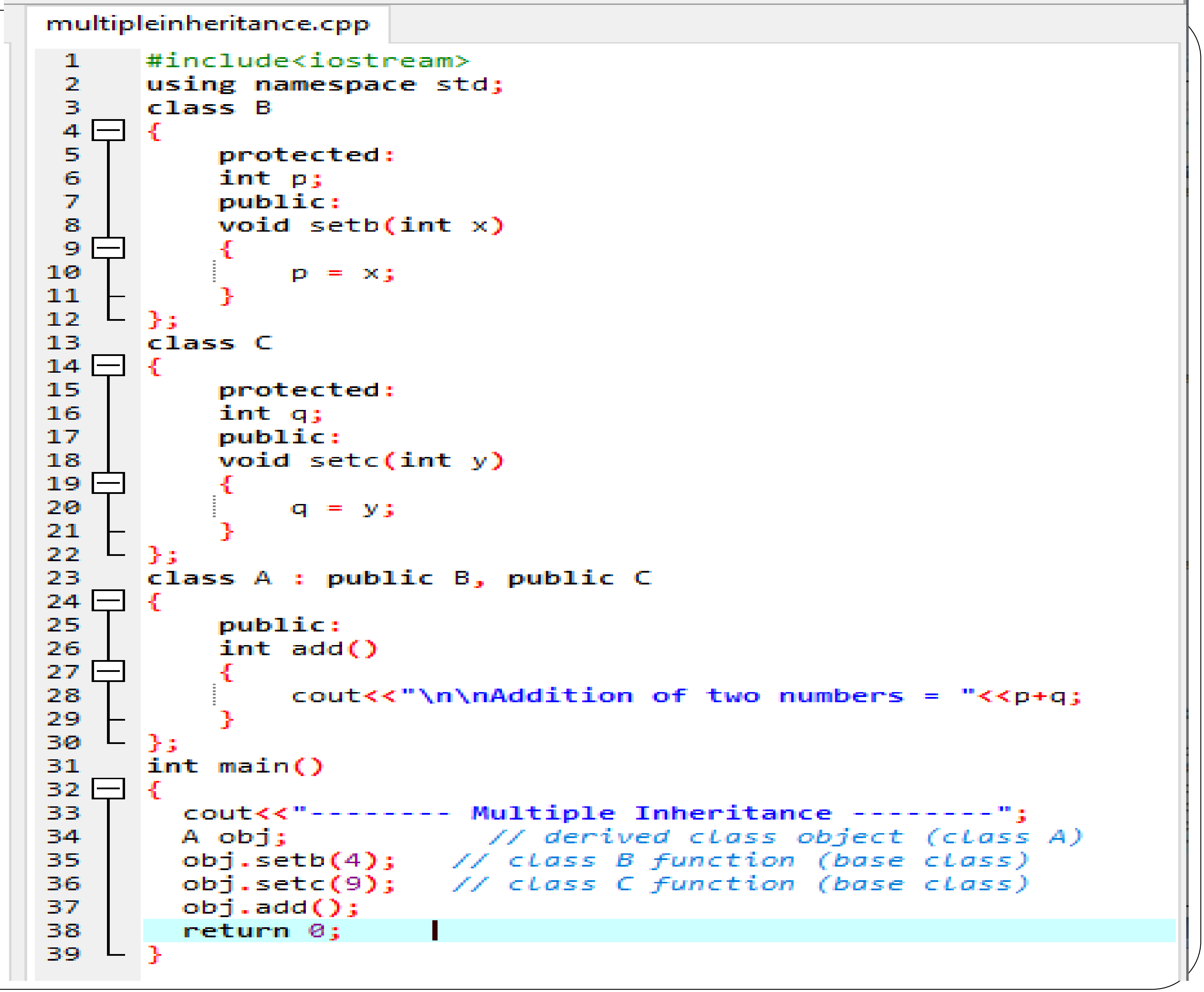
- In this type of inheritance, **one** derived class inherits from **more than one** base class.
- A **derived class** with **several base classes** is called as **Multiple Inheritance**



- Syntax :

```
class base1
{
   ..........//body
};
class base2
{
   ...........//body
};
.
.
.
class derived : access_specifier base1, access_specifier base2,
...
{
           //body of the derived class
};
```

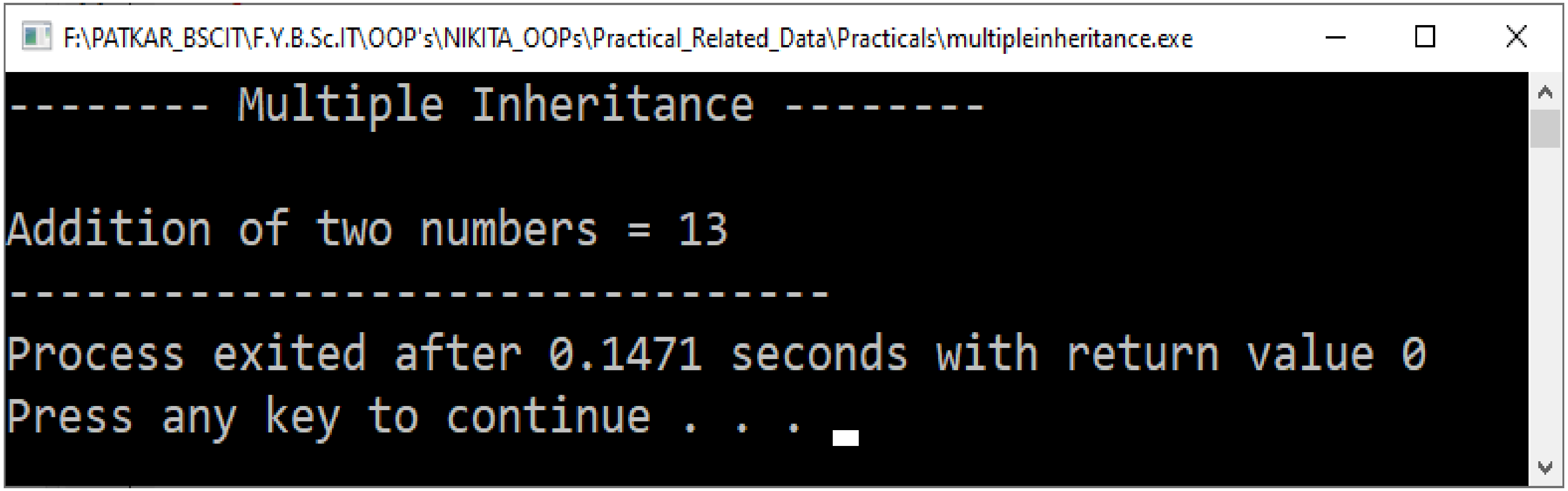
e.g.



multipleinheritance.cpp

```cpp
#include<iostream>
using namespace std;
class B
{
    protected:
    int p;
    public:
    void setb(int x)
    {
        p = x;
    }
};
class C
{
    protected:
    int q;
    public:
    void setc(int y)
    {
        q = y;
    }
};
class A : public B, public C
{
    public:
    int add()
    {
        cout<<"\n\nAddition of two numbers = "<<p+q;
    }
};
int main()
{
  cout<<"--------- Multiple Inheritance ---------";
  A obj;           // derived class object (class A)
  obj.setb(4);   // class B function (base class)
  obj.setc(9);   // class C function (base class)
  obj.add();
  return 0;
}
```

Output :

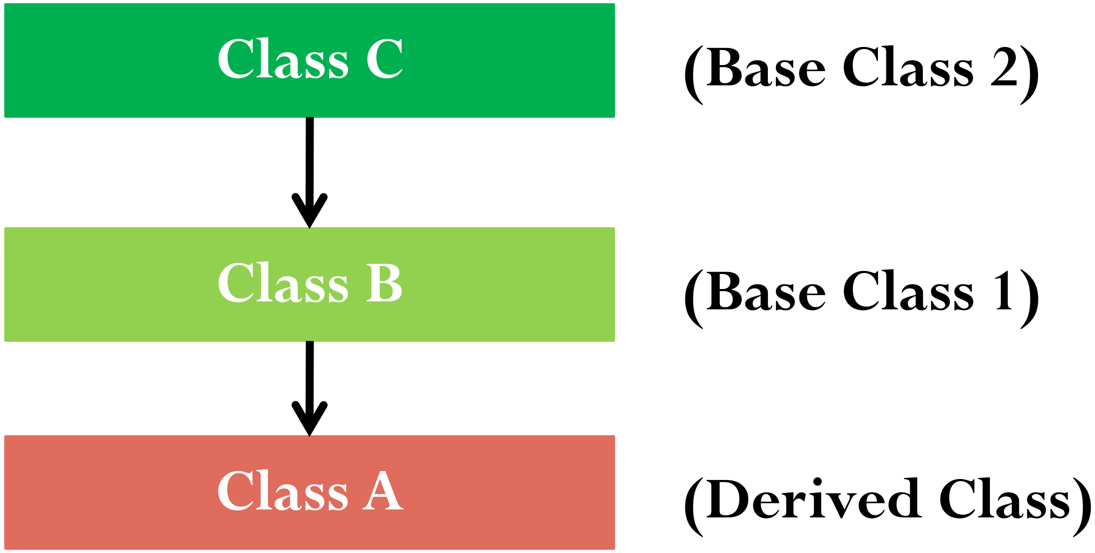


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\multipleinheritance.exe

```
-------- Multiple Inheritance --------

Addition of two numbers = 13
--------------------------------
Process exited after 0.1471 seconds with return value 0
Press any key to continue . . .
```

Multilevel Inheritance

- The mechanism of deriving a **class from another** ‘derived class’ is known as **multilevel inheritance**
- In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other



Class C (Base Class 2)

Class B (Base Class 1)

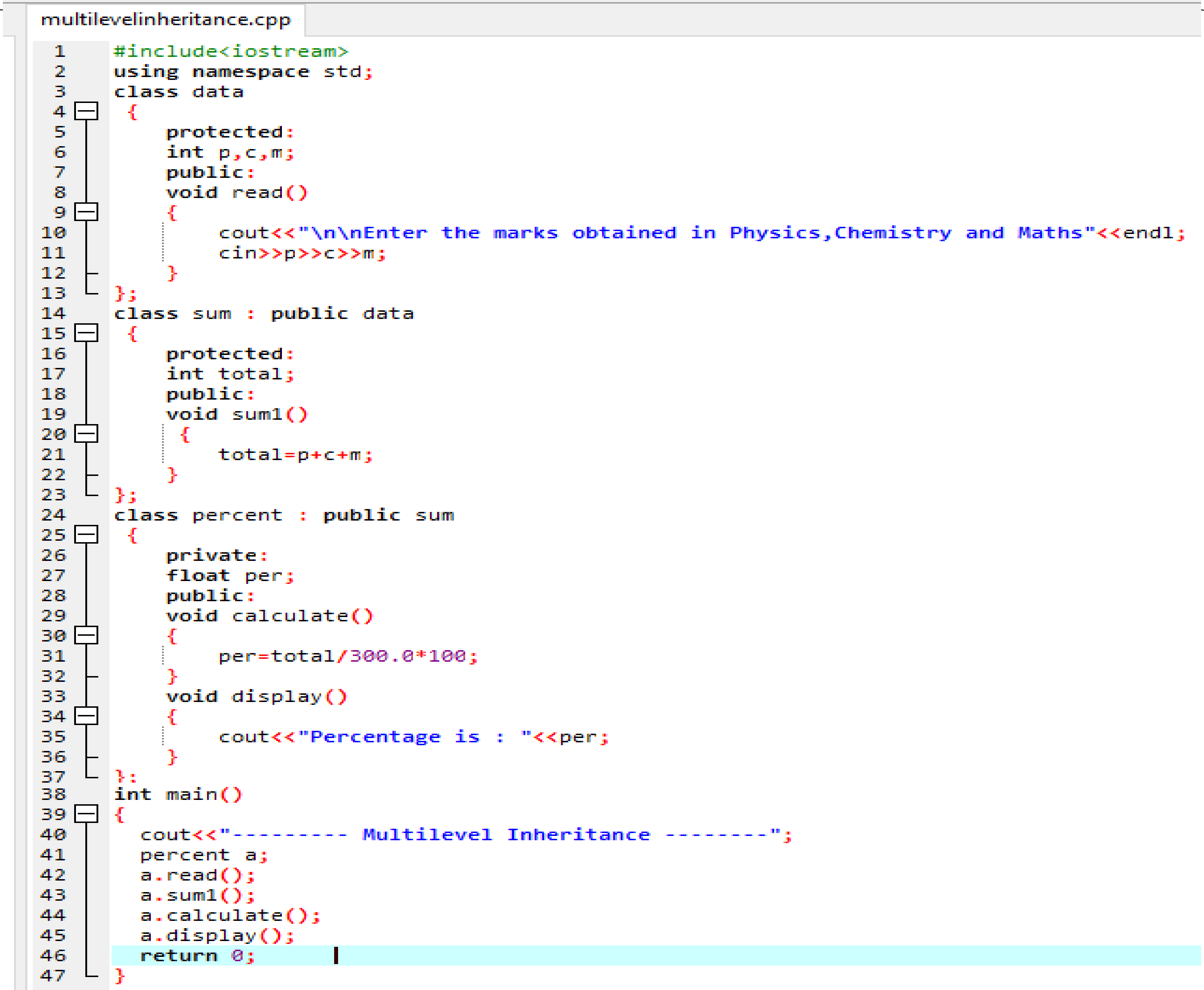
Class A (Derived Class)

The class C serves as a base class for the derived class B, which in turn serves as a base class for the derived class A. The class B is known as **intermediate base** class since it provides a link for the inheritance between C and A

- Syntax :

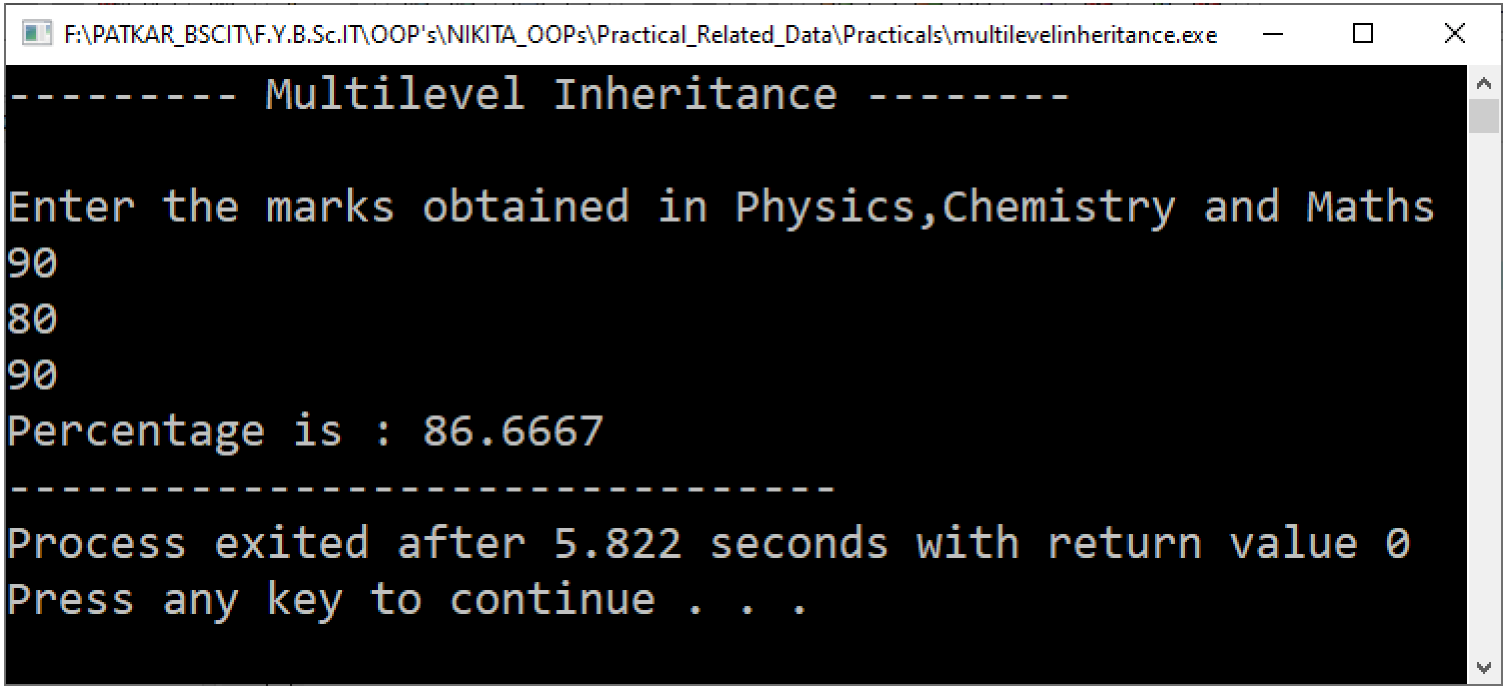
```
class C                //Base Class
{
   ....... //body
};
class B : access_specifier C     //Derived from C
{
  ........//body
};
class A : access_specifier B     //Derived from B
 {
  .........//body
};
```

e.g.



```cpp
#include<iostream>
using namespace std;
class data
 {
    protected:
    int p,c,m;
    public:
    void read()
    {
        cout<<"\n\nEnter the marks obtained in Physics,Chemistry and Maths"<<endl;
        cin>>p>>c>>m;
    }
};
class sum : public data
 {
    protected:
    int total;
    public:
    void sum1()
     {
        total=p+c+m;
    }
};
class percent : public sum
 {
    private:
    float per;
    public:
    void calculate()
    {
        per=total/300.0*100;
    }
    void display()
    {
        cout<<"Percentage is : "<<per;
    }
};
int main()
{
  cout<<"--------- Multilevel Inheritance --------";
  percent a;
  a.read();
  a.sum1();
  a.calculate();
  a.display();
  return 0;
}
```

Output :

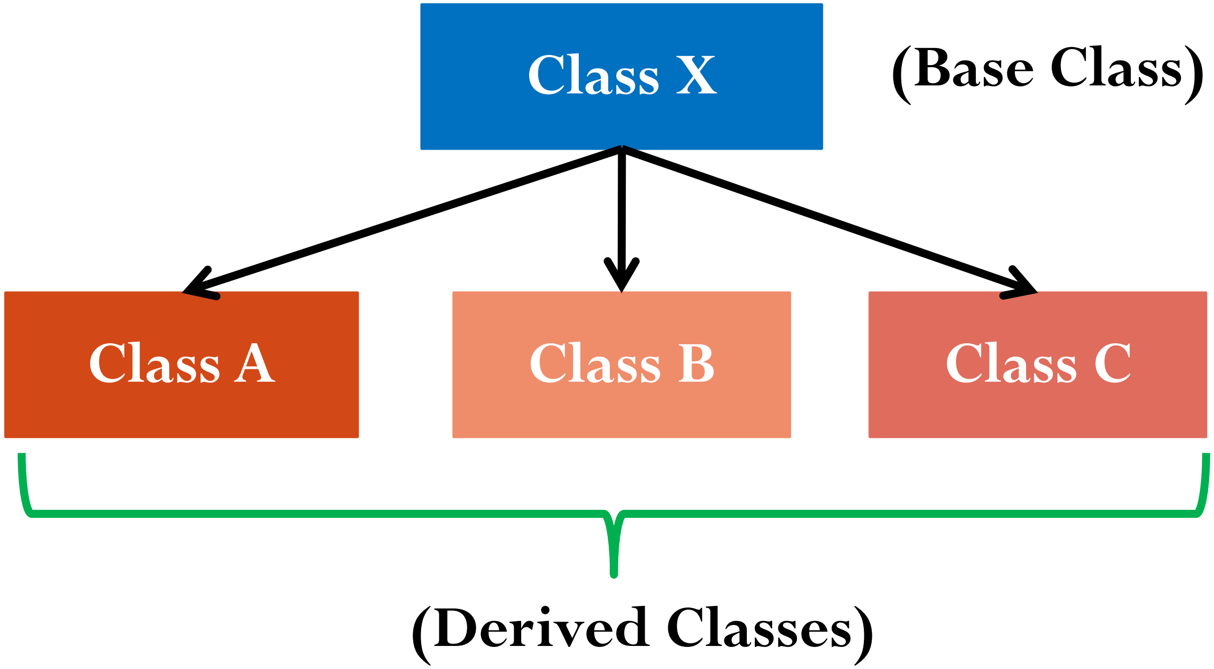


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\multilevelinheritance.exe

```
--------- Multilevel Inheritance ---------

Enter the marks obtained in Physics,Chemistry and Maths
90
80
90
Percentage is : 86.6667
--------------------------------
Process exited after 5.822 seconds with return value 0
Press any key to continue . . .
```

Hierarchical Inheritance

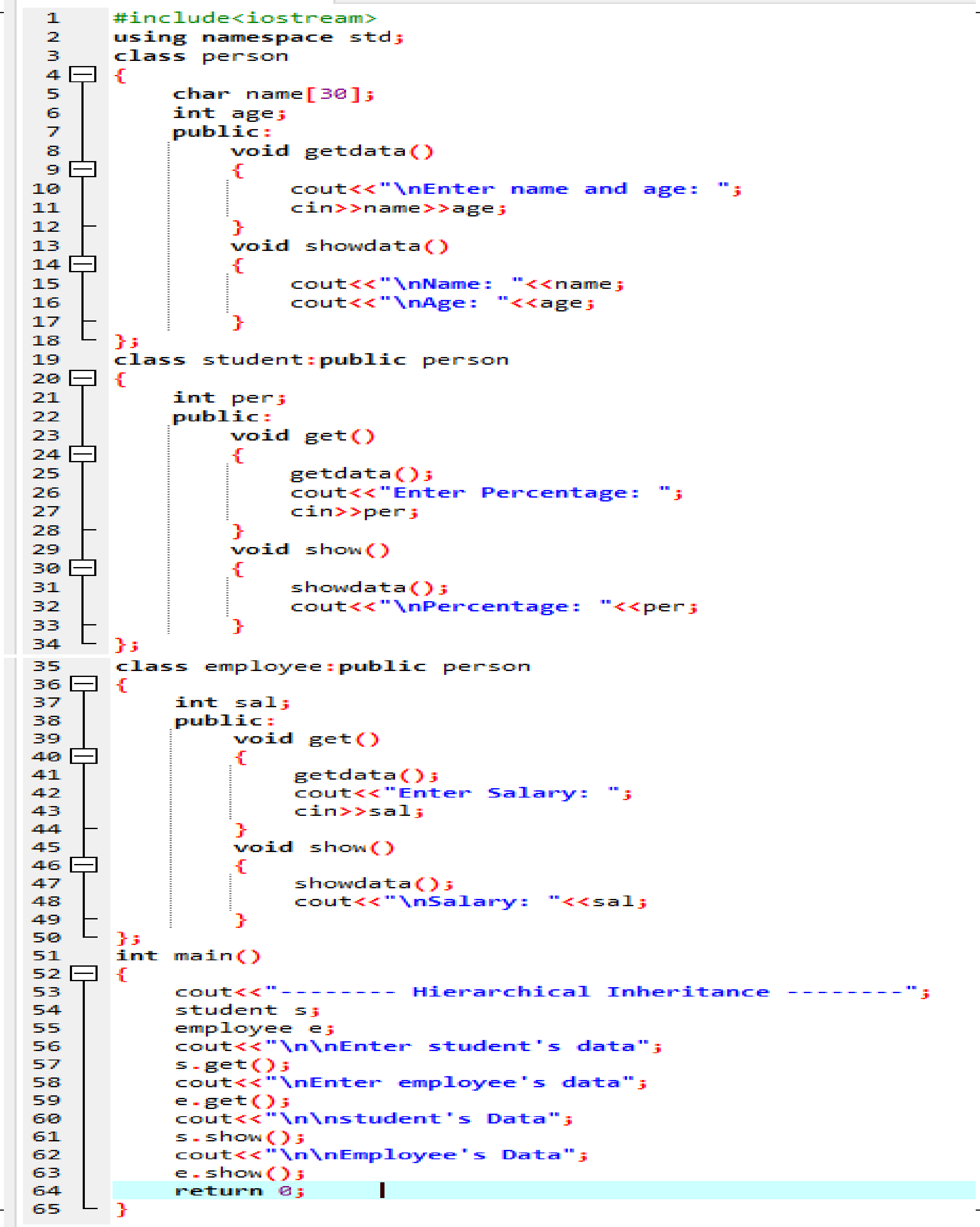
- **One class** may be inherited by **more than one class**. This process is known as **Hierarchical Inheritance**
- In this type of inheritance, **multiple** derived classes inherits from a **single** base class



- Syntax :

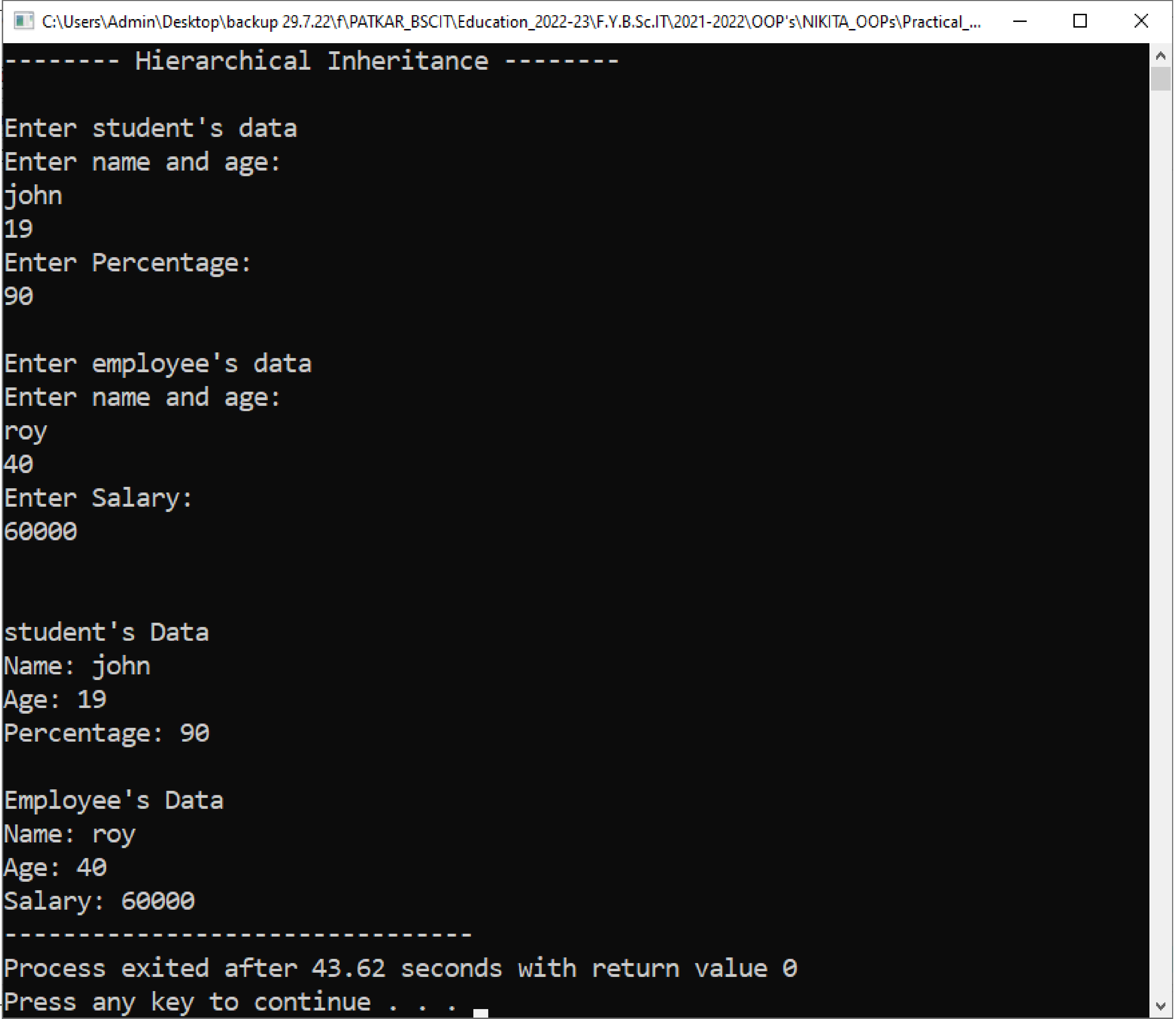
```
class base
 {
            //body
};
class derived1 : access_specifier base
{
            //body
};
class derived2 : access_specifier base
{
            //body
};
```

e.g.



```
#include<iostream>
using namespace std;
class person
{
    char name[30];
    int age;
    public:
        void getdata()
        {
            cout<<"\nEnter name and age: ";
            cin>>name>>age;
        }
        void showdata()
        {
            cout<<"\nName: "<<name;
            cout<<"\nAge: "<<age;
        }
};
class student:public person
{
    int per;
    public:
        void get()
        {
            getdata();
            cout<<"Enter Percentage: ";
            cin>>per;
        }
        void show()
        {
            showdata();
            cout<<"\nPercentage: "<<per;
        }
};
class employee:public person
{
    int sal;
    public:
        void get()
        {
            getdata();
            cout<<"Enter Salary: ";
            cin>>sal;
        }
        void show()
        {
            showdata();
            cout<<"\nSalary: "<<sal;
        }
};
int main()
{
    cout<<"-------- Hierarchical Inheritance --------";
    student s;
    employee e;
    cout<<"\n\nEnter student's data";
    s.get();
    cout<<"\nEnter employee's data";
    e.get();
    cout<<"\n\nstudent's Data";
    s.show();
    cout<<"\n\nEmployee's Data";
    e.show();
    return 0;
}
```

Output :



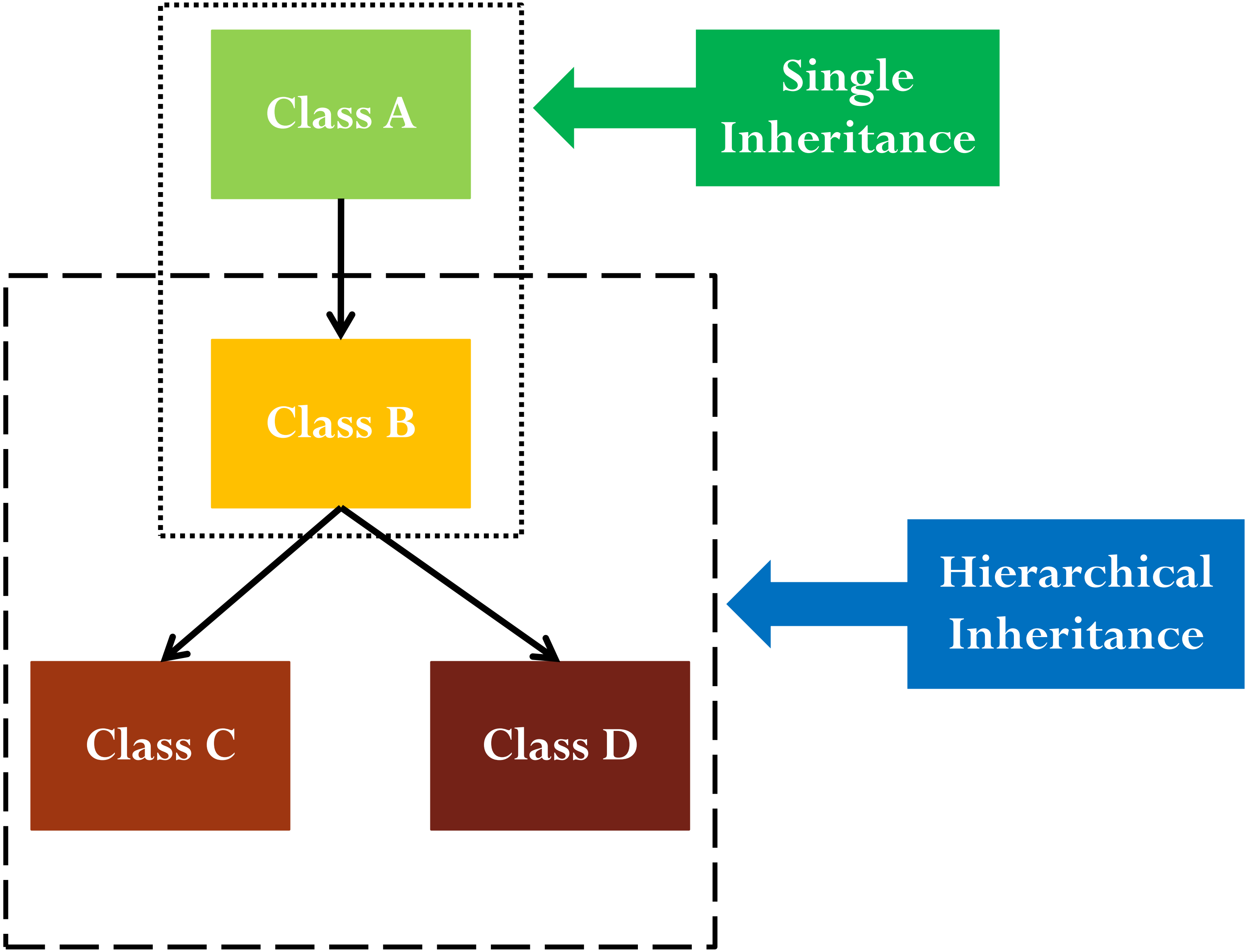
C:\Users\Admin\Desktop\backup 29.7.22\f\PATKAR_BSCIT\Education_2022-23\F.Y.B.Sc.IT\2021-2022\OOP's\NIKITA_OOPs\Practical_...

```
-------- Hierarchical Inheritance --------

Enter student's data
Enter name and age:
john
19
Enter Percentage:
90

Enter employee's data
Enter name and age:
roy
40
Enter Salary:
60000


student's Data
Name: john
Age: 19
Percentage: 90

Employee's Data
Name: roy
Age: 40
Salary: 60000
--------------------------------
Process exited after 43.62 seconds with return value 0
Press any key to continue . . .
```

Hybrid Inheritance

- **Hybrid inheritance** is the **combination** of two or more **types of inheritance**
- Various combinations can be make in hybrid inheritance

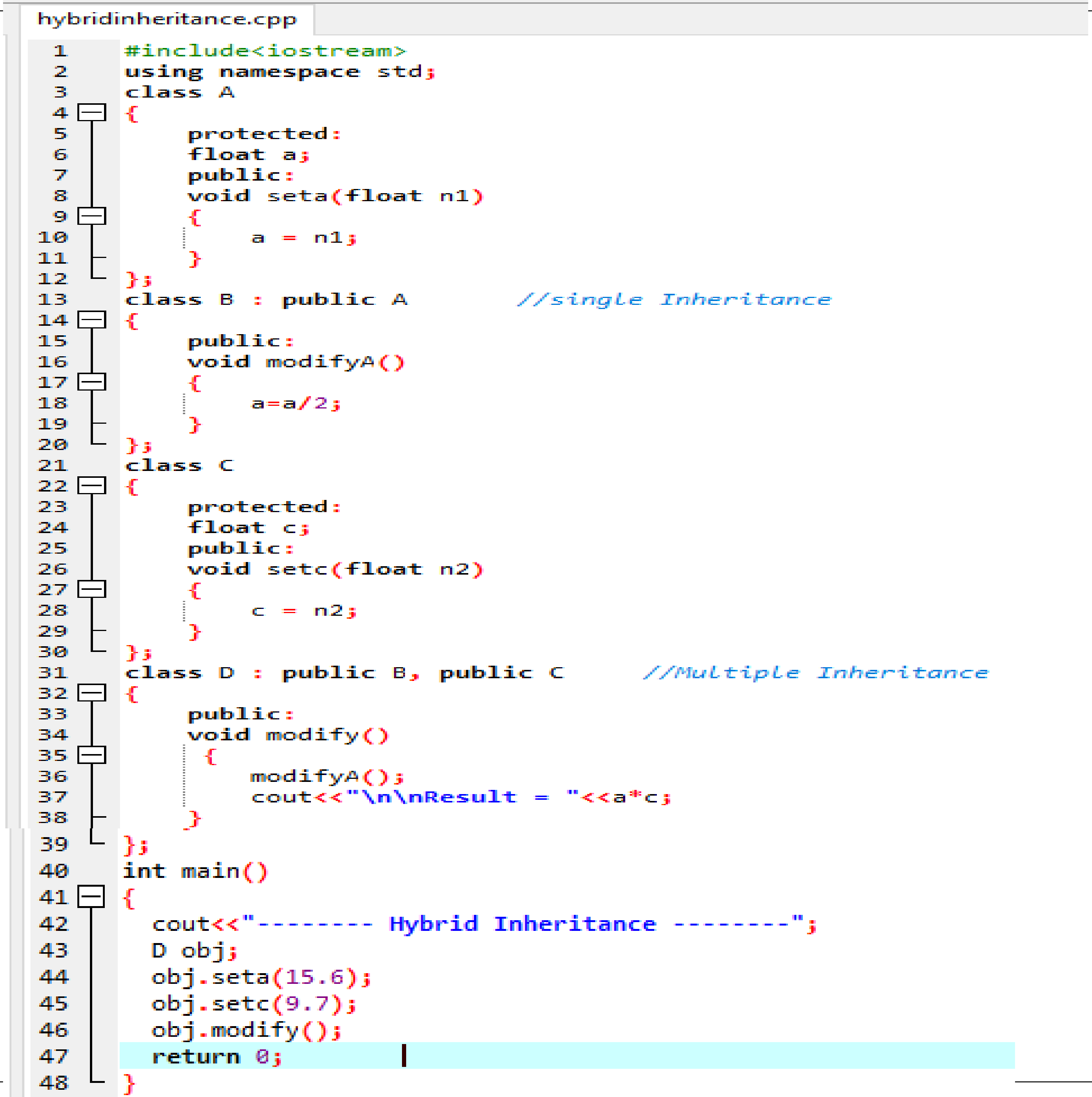
Fig.


Class A
Single Inheritance
Class B
Hierarchical Inheritance
Class C
Class D

Syntax

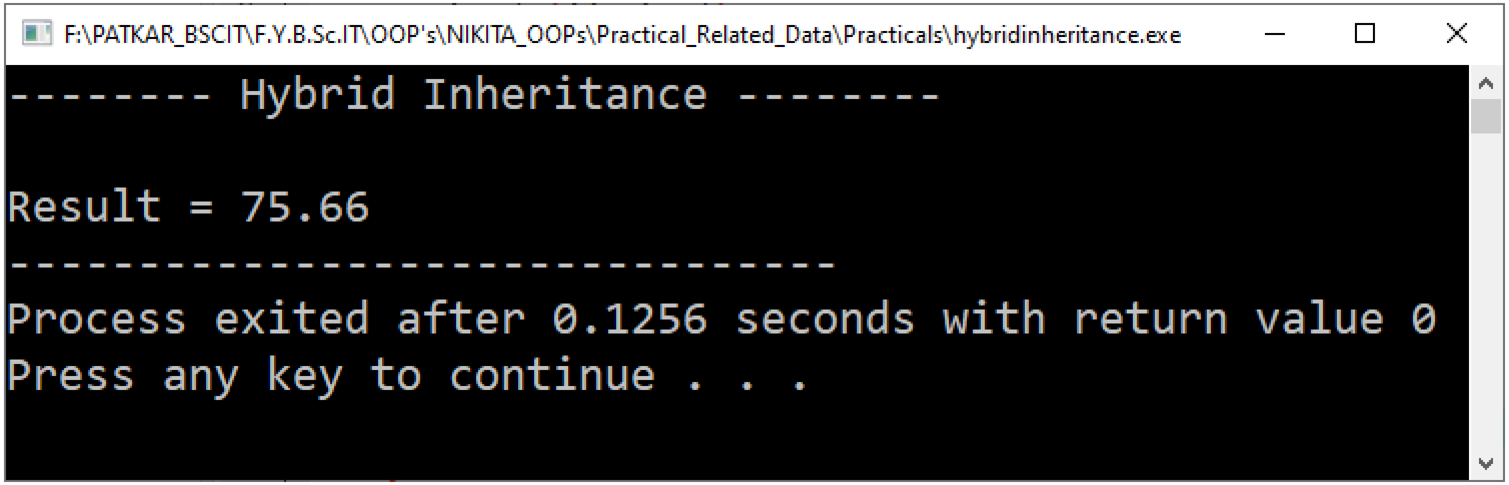
```
class A
{
     .........
 };
class B : public A       //Single Inheritance
{
    ..........
} ;
class C
{
    ...........
 };
 class D : public B, public C    //Multiple Inheritance
{
    ...........
 };
```

e.g.



```cpp
#include<iostream>
using namespace std;
class A
{
	protected:
	float a;
	public:
	void seta(float n1)
	{
		a = n1;
	}
};
class B : public A		//single Inheritance
{
	public:
	void modifyA()
	{
		a=a/2;
	}
};
class C
{
	protected:
	float c;
	public:
	void setc(float n2)
	{
		c = n2;
	}
};
class D : public B, public C		//Multiple Inheritance
{
	public:
	void modify()
	 {
		modifyA();
		cout<<"\n\nResult = "<<a*c;
	}
};
int main()
{
   cout<<"-------- Hybrid Inheritance --------";
   D obj;
   obj.seta(15.6);
   obj.setc(9.7);
   obj.modify();
   return 0;
}
```

Output :



F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\hybridinheritance.exe

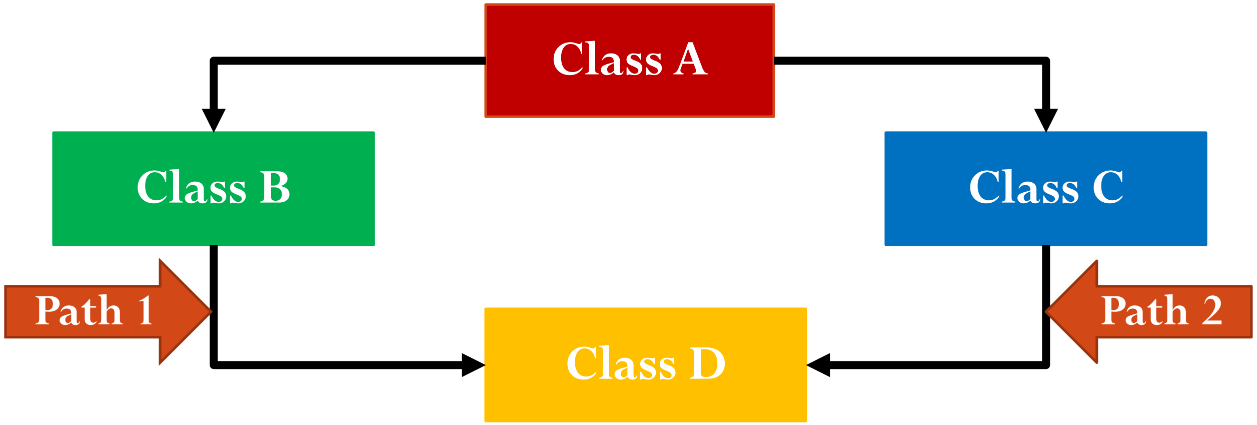
```
-------- Hybrid Inheritance --------

Result = 75.66
--------------------------------
Process exited after 0.1256 seconds with return value 0
Press any key to continue . . .
```

Special Case of Hybrid Inheritance (Multipath Inheritance)

- When a class is derived from two or more classes, which are derived from the same base class such type of inheritance is known as **multipath inheritance** (i.e. **A derived class with two or more base classes and these two base classes have one common base class is called multipath inheritance**)
- **Multipath inheritance** consists **multiple**, **multilevel** **and** **hierarchical** as shown in the figure

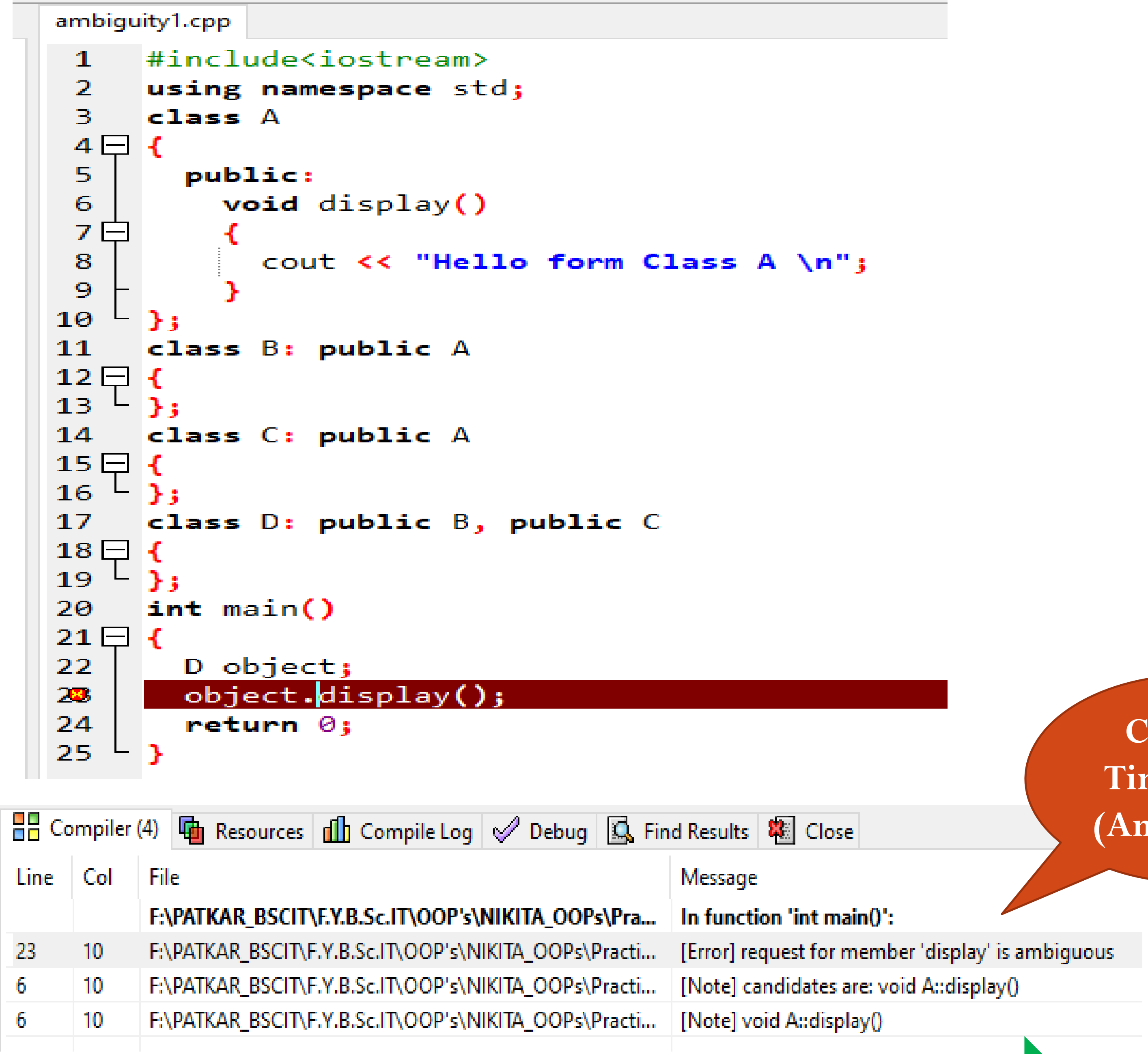
Next Slide

Fig.



- **Explanation of diagram**
- Class A is the **parent class**, and **Classes B** and **C** are the **derived classes** from **Class A**. Thus **Class B** and **Class C** have all the **properties of Class A**.
- Next, **Class D inherits Class B** and also inherits **Class C**. With this Class D will **get** all the **properties from** Class B and **Class C** which **also** has **Class A's properties in them.**
- What do you think will happen when we will try to access Class A's properties through D? There will be an **error** because **Class D will get Class A's properties twice and the compiler couldn't decide what to execute and thrwos an error**. You'll see the term **"ambiguous"** when such a situation occurs
- An ambiguity (duplicity) can arise in this type of inheritance
- When you run a program with such type of inheritance. It gives a compile time error [Ambiguity].

e.g.



```
1   #include<iostream>
2   using namespace std;
3   class A
4   {
5      public:
6         void display()
7         {
8            cout << "Hello form Class A \n";
9         }
10  };
11  class B: public A
12  {
13  };
14  class C: public A
15  {
16  };
17  class D: public B, public C
18  {
19  };
20  int main()
21  {
22     D object;
23     object.display();
24     return 0;
25  }
```

```
Line  Col  File                                              Message
           F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Pra...  In function 'int main()':
23    10   F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Error] request for member 'display' is ambiguous
6     10   F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Note] candidates are: void A::display()
6     10   F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practi...  [Note] void A::display()
```

Compile Time Error (Ambiguity)

Solution for this is Next Slide

Virtual Base Class

- To **overcome** the **ambiguity** occurred due to multipath inheritance, C++ **introduced virtual base class**
- When a class is made **virtual**, **necessary care** is **taken** so that the **duplication** is **avoided** regardless of the number of paths that exist to the child class
- When classes are declared as **virtual**, the **compiler** takes the necessary **precaution** to **avoid duplication** of data members. Only **one copy of** its **data members** is **shared** by all the **base classes** that use virtual base class

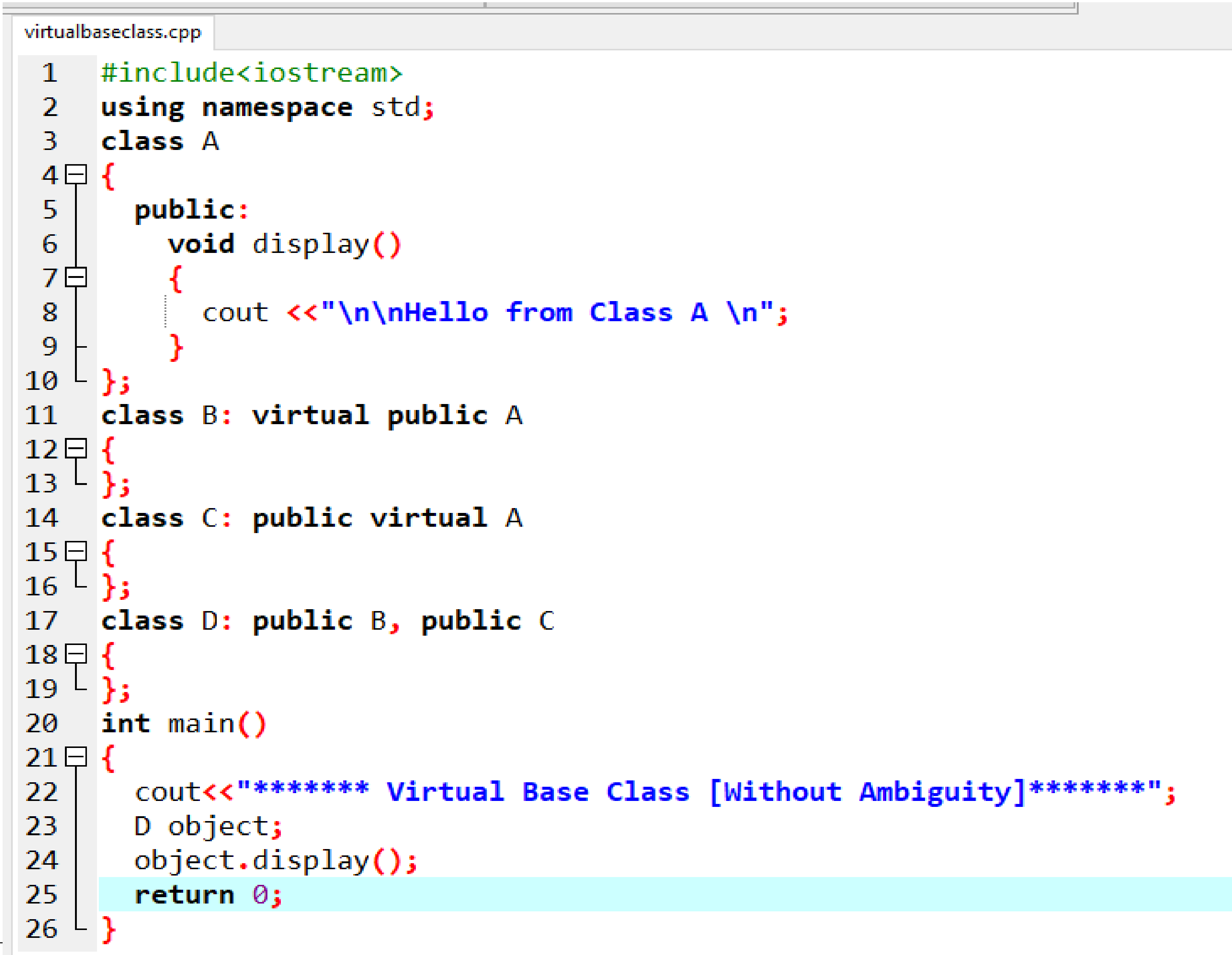
- Virtual base classes in C++ are used in a way of preventing multiple instances (copy) of a given class appearing in an inheritance hierarchy when using multipath inheritance
- If a virtual base class is **not used**, then all the derived classes will **get duplicated data members**. In this case, the **compiler** **cannot decide** which **one to execute**
- The keyword **virtual** declares the specified class **virtual**
- **By adding virtual keyword compiler will automatically will decide the path**

- **Syntax**
- If Class A is considered as the base class and Class B and Class C are considered as the derived classes of A.

```
class A
{
   //code
};
class B:  virtual public A
{
   //code
};
class C: public virtual A
{
   //code
 };
```

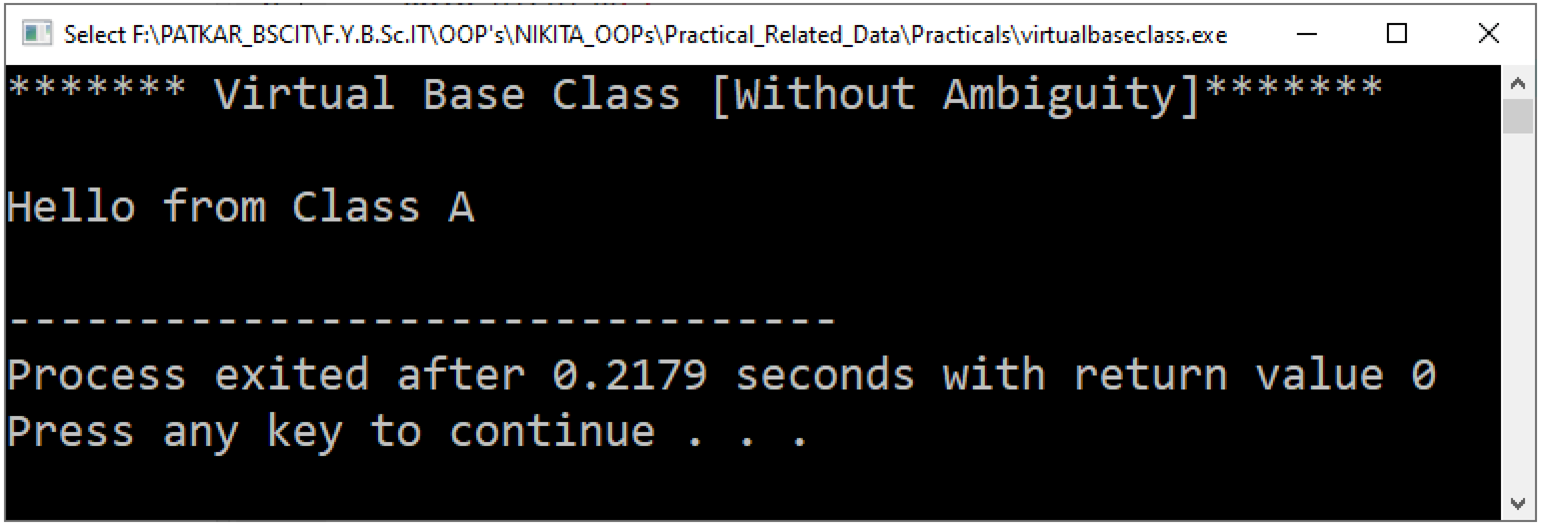
- **Note :**

The word **“virtual”** can be written **before or after** the word **“public”** e.g. **Ambiguity error resolved after using Virtual Base Class**



```cpp
#include<iostream>
using namespace std;
class A
{
  public:
    void display()
    {
      cout <<"\n\nHello from Class A \n";
    }
};
class B: virtual public A
{
};
class C: public virtual A
{
};
class D: public B, public C
{
};
int main()
{
  cout<<"******* Virtual Base Class [Without Ambiguity]*******";
  D object;
  object.display();
  return 0;
}
```

Output:



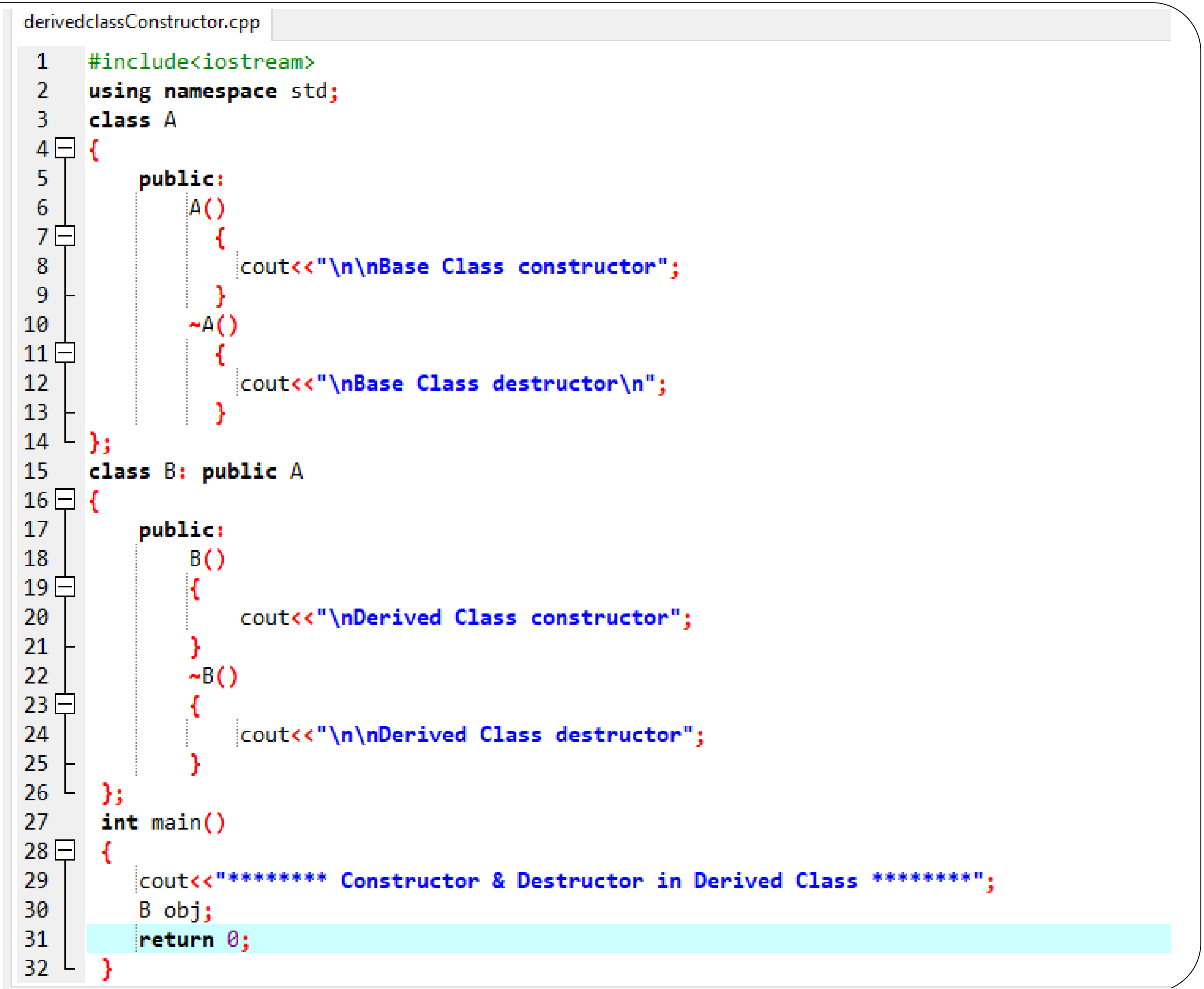
Select F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\virtualbaseclass.exe

```
******* Virtual Base Class [Without Ambiguity]*******

Hello from Class A


--------------------------------
Process exited after 0.2179 seconds with return value 0
Press any key to continue . . .
```

Constructor and Destructor in Inheritance

- In inheritance, When an object of derived class is created then **constructor of base class** is executed **first** and then it executed the **constructor of derived class**.
- Similarly, the **destructor are executed in *reverse order***, i.e. When an object of derived class is created then ***destructor of derived class is executed first*** and then it calls the destructor of base class.

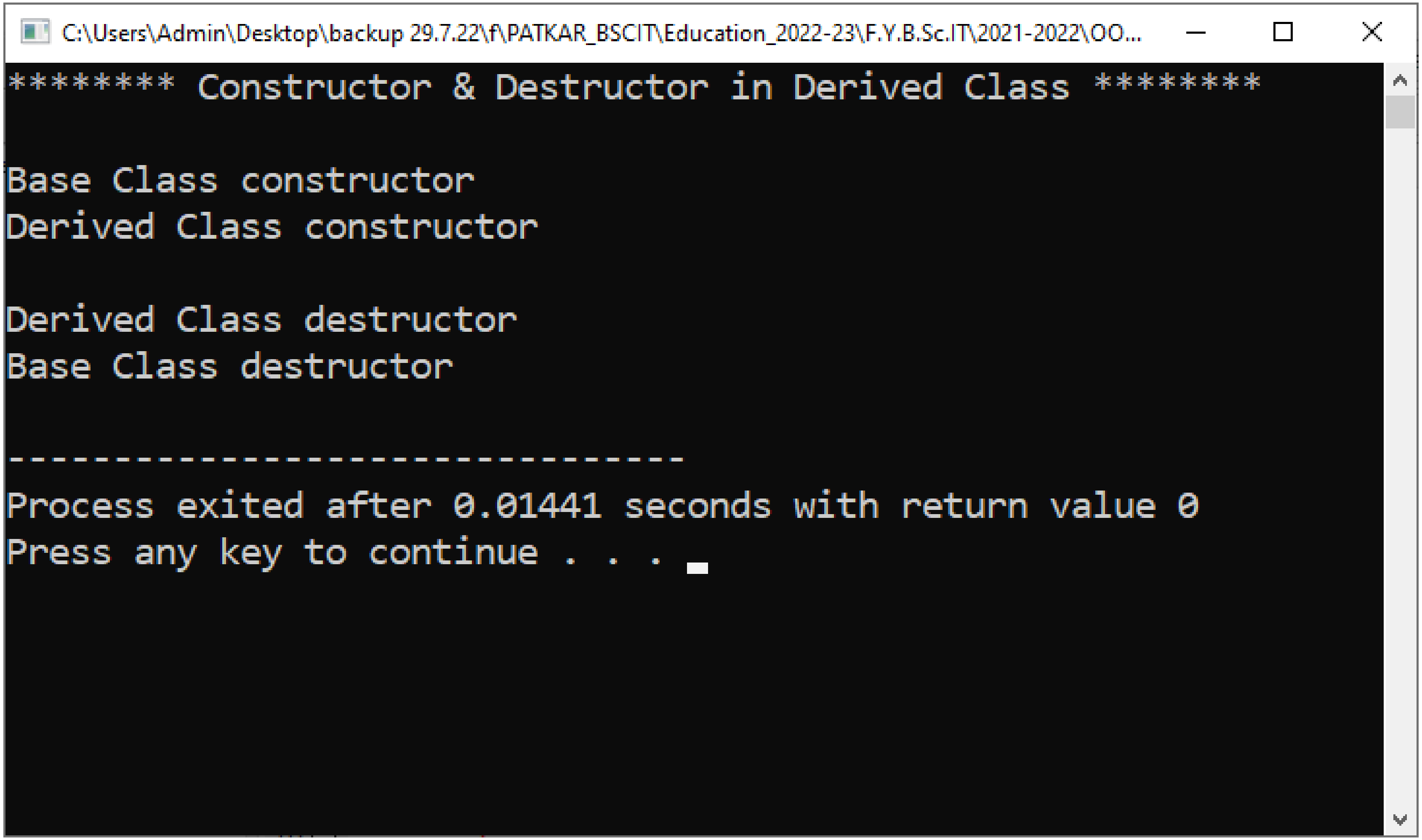
e.g.



derivedclassConstructor.cpp

```cpp
#include<iostream>
using namespace std;
class A
{
    public:
        A()
          {
            cout<<"\n\nBase Class constructor";
          }
        ~A()
          {
            cout<<"\nBase Class destructor\n";
          }
};
class B: public A
{
    public:
        B()
        {
            cout<<"\nDerived Class constructor";
        }
        ~B()
        {
            cout<<"\n\nDerived Class destructor";
        }
 };
 int main()
 {
    cout<<"******** Constructor & Destructor in Derived Class ********";
    B obj;
    return 0;
 }
```

Output:



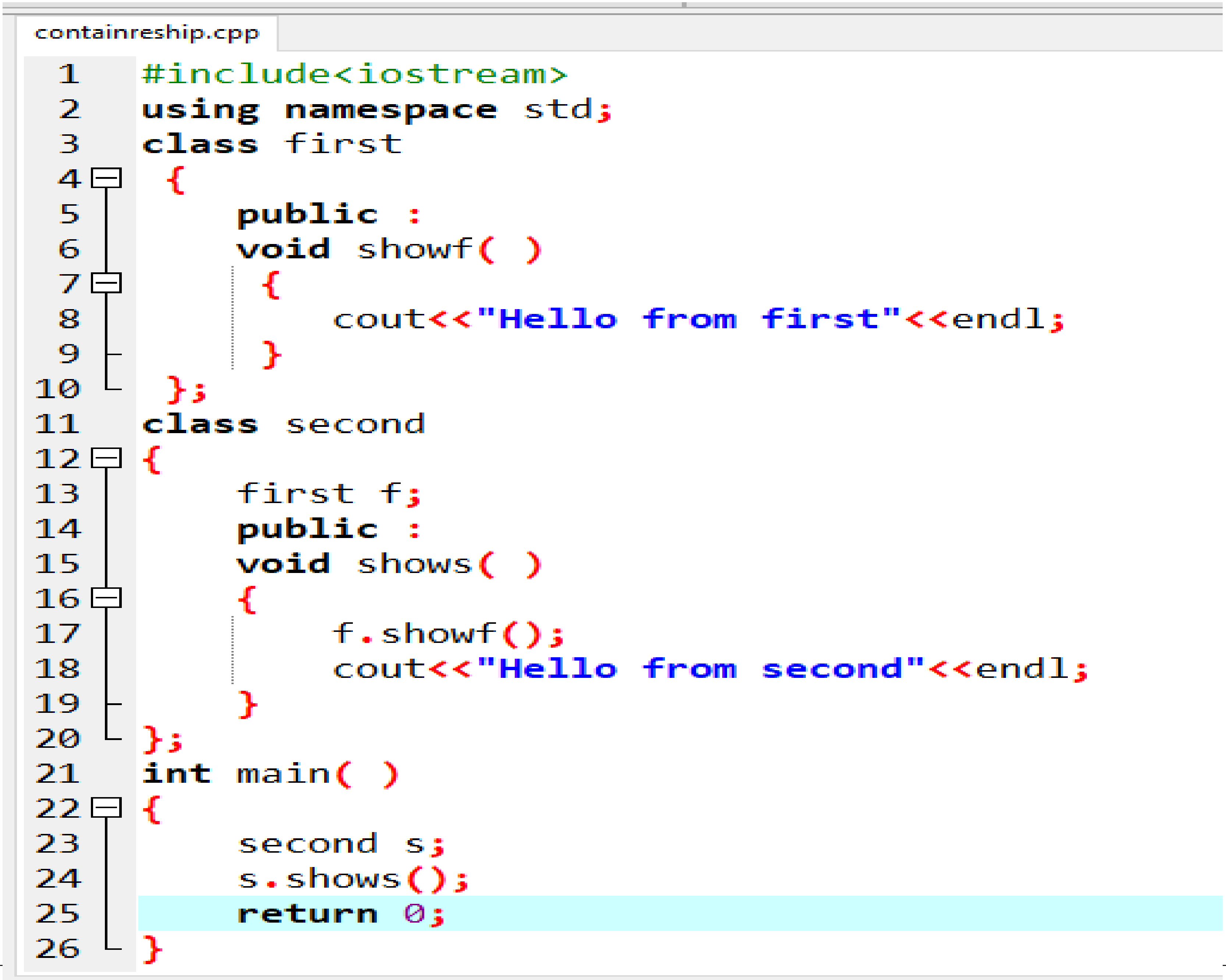
C:\Users\Admin\Desktop\backup 29.7.22\f\PATKAR_BSCIT\Education_2022-23\F.Y.B.Sc.IT\2021-2022\OO...

```
******** Constructor & Destructor in Derived Class ********

Base Class constructor
Derived Class constructor

Derived Class destructor
Base Class destructor


--------------------------------
Process exited after 0.01441 seconds with return value 0
Press any key to continue . . .
```

Containership

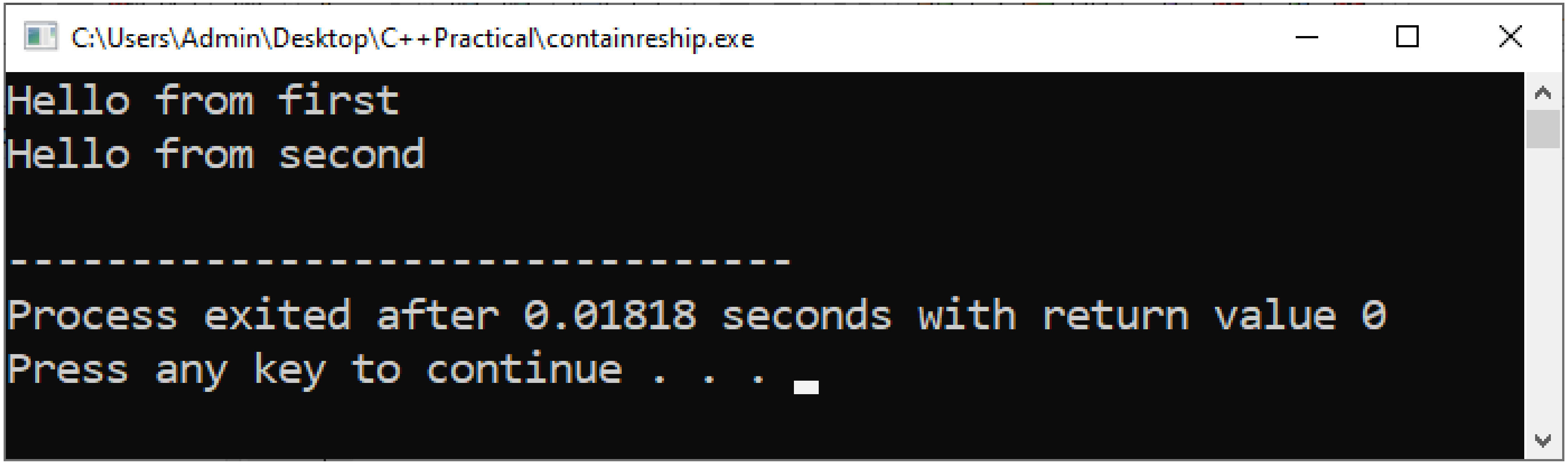
- When an **object of one class** is **created** into **another class** then that object will be a member of that class, this **type of relationship** between the classes is **known as Containership** or **has_a** relationship as one class contains the object of another class.
- The class which **contains the object** and **member of another class** that class is **called** as **container class** and the ***object that is part of another object*** is called a ***contained object*** whereas the *object that contains another object* as its part is called a *container object*.

- E.g



```cpp
#include<iostream>
using namespace std;
class first
 {
    public :
    void showf( )
     {
        cout<<"Hello from first"<<endl;
     }
 };
class second
{
    first f;
    public :
    void shows( )
    {
        f.showf();
        cout<<"Hello from second"<<endl;
    }
};
int main( )
{
    second s;
    s.shows();
    return 0;
}
```

- Output:



```
Hello from first
Hello from second


--------------------------------
Process exited after 0.01818 seconds with return value 0
Press any key to continue . . .
```

10.
POINTERS TO OBJECTS AND VIRTUAL FUNCTIONS

Pointer to Objects

- A **variable that holds an address** value is called a **pointer variable or simply pointer**.
- Pointers are used to store address of variables which have data types like int, float, double, etc.
- But **pointer can also store the address of an object**.
- Declaration of pointer variable as follows:

classname *obj_pointer;

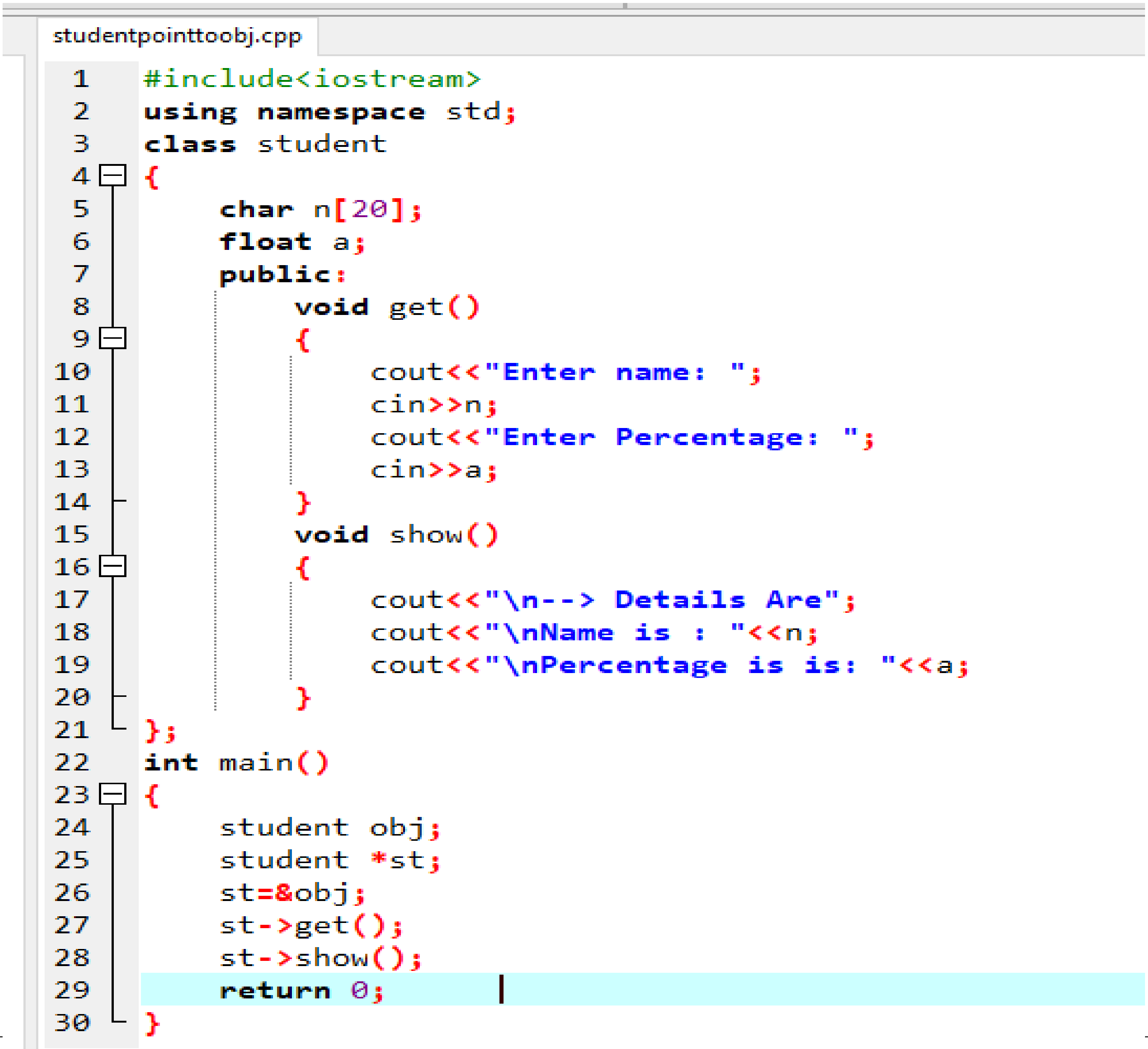
- **Storing** the address of an object in the pointer variable. To find the **address of an object**, place the **"&"operator** before the object's name as follows:

obj_pointer = &obj_name;

- When **accessing members of a class** using an object pointer, the **arrow operator (->)** is used instead of dot operator.
- To access the members of a class using a pointer to that class, this ” ->” operator must use as follows:

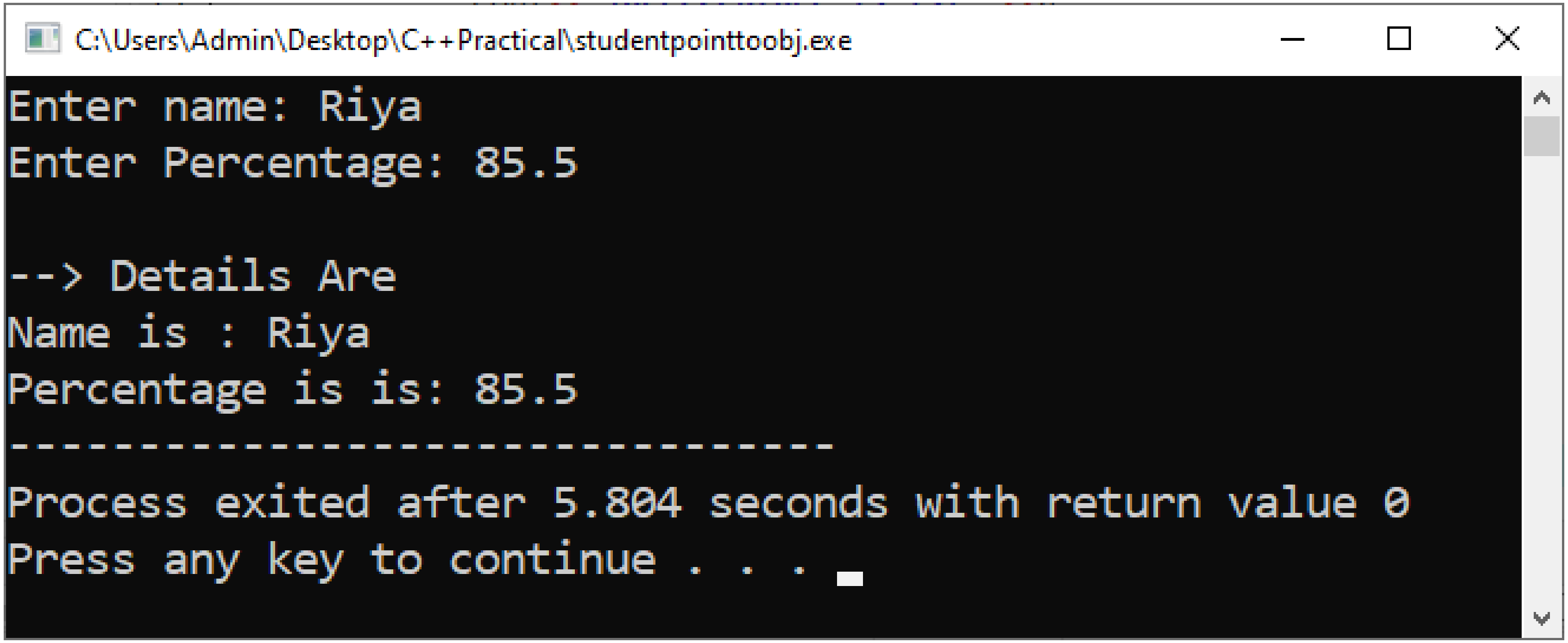
obj_pointer->function_name;

- E.g.



```cpp
#include<iostream>
using namespace std;
class student
{
    char n[20];
    float a;
    public:
        void get()
        {
            cout<<"Enter name: ";
            cin>>n;
            cout<<"Enter Percentage: ";
            cin>>a;
        }
        void show()
        {
            cout<<"\n--> Details Are";
            cout<<"\nName is : "<<n;
            cout<<"\nPercentage is is: "<<a;
        }
};
int main()
{
    student obj;
    student *st;
    st=&obj;
    st->get();
    st->show();
    return 0;
}
```

- Output:



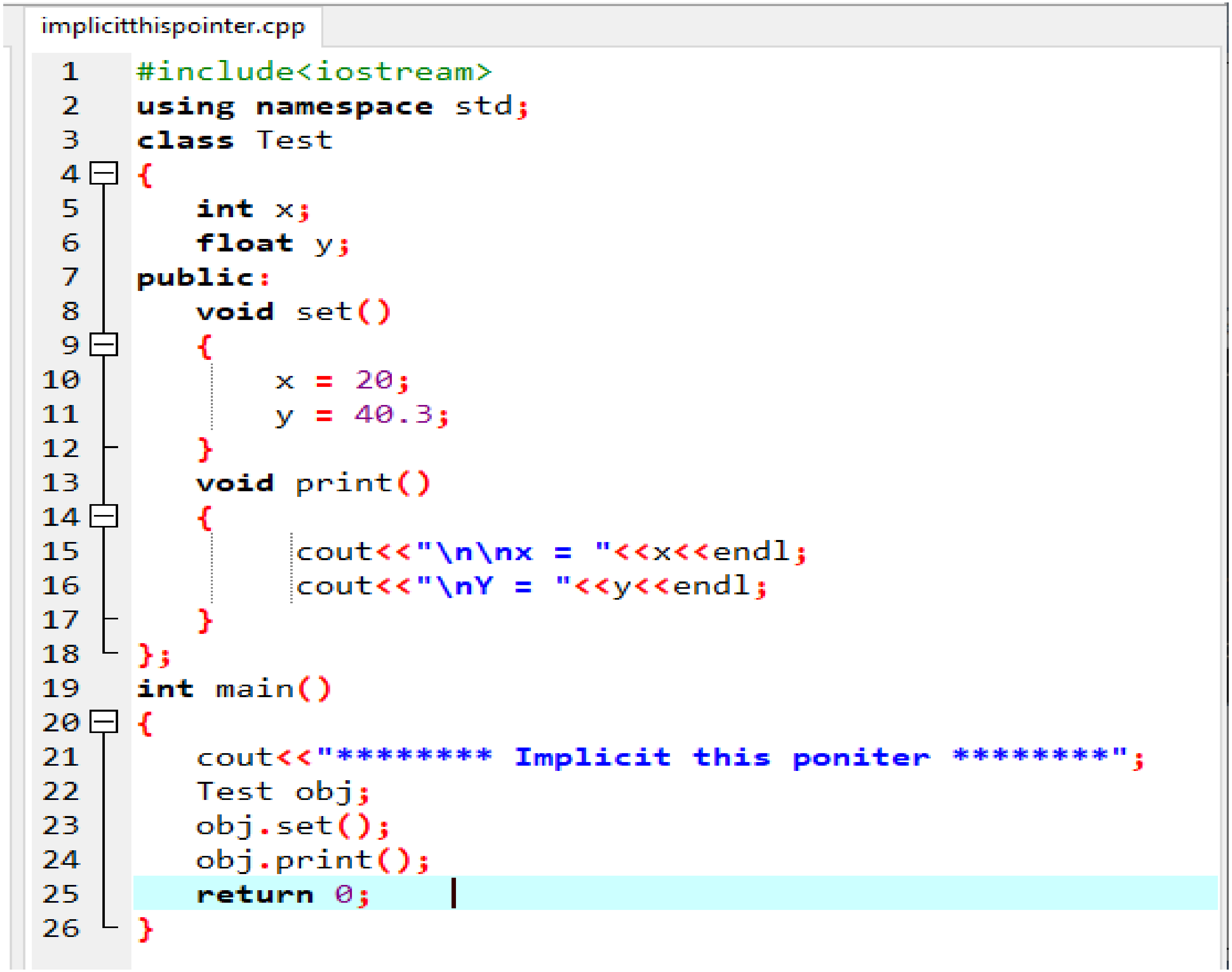
```
Enter name: Riya
Enter Percentage: 85.5

--> Details Are
Name is : Riya
Percentage is is: 85.5
--------------------------------
Process exited after 5.804 seconds with return value 0
Press any key to continue . . .
```

The this Pointer

- The **this pointer** holds the **address** of **current object** (i.e. pointer points to the current object of the class)
- Every object in C++ has access to its own address through an important pointer called **this pointer**
- **this pointer** represents an object that invoke or call a member function
- **this pointer** is automatically passed to a member function when it is called (When member function is called it automatically passes a pointer to invoking object)

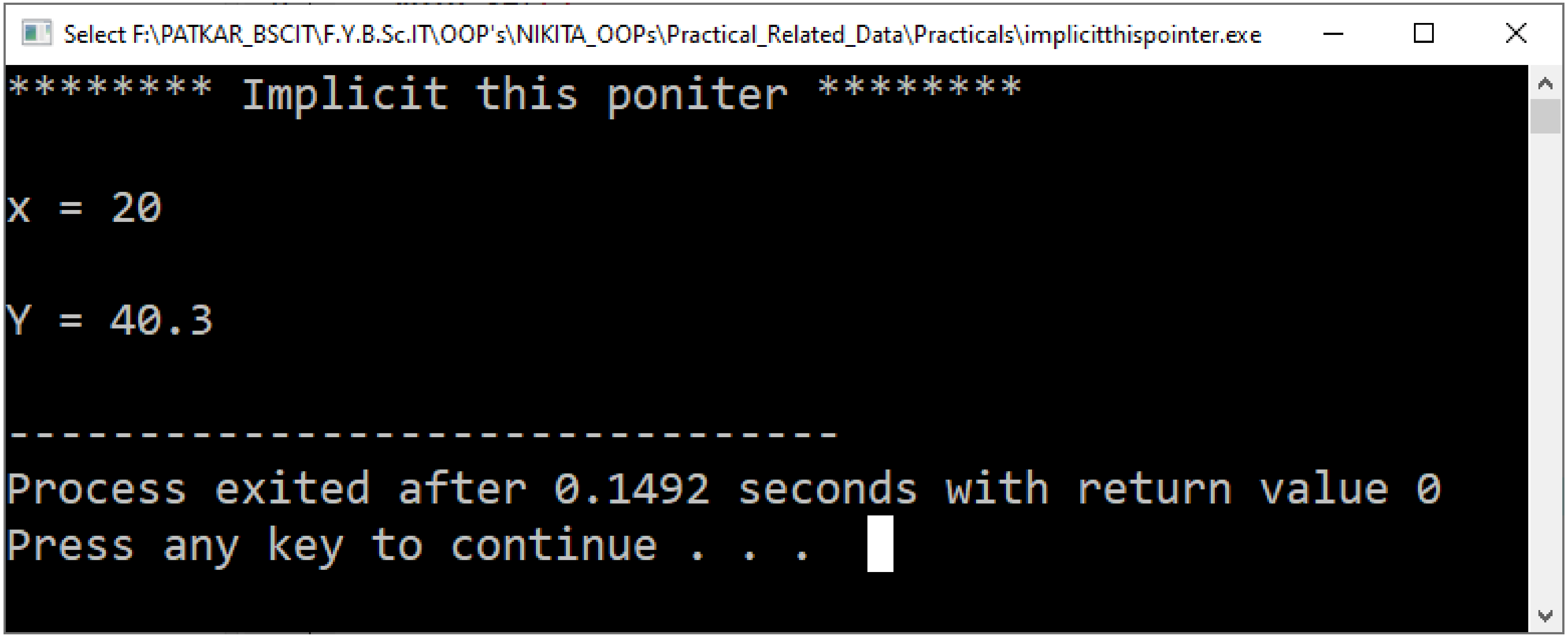
- The **this pointer** is an **implicit parameter** to all member functions.
- Therefore, **inside a member function**, this may be **used** to **refer** to the invoking object.
- **this pointer is use when local variable is same as member name**
- **Friend functions** do not have a **this pointer**, because friends are not members of a class.
- Only **member functions** have a **this pointer**

- E.g. Implicit Passing pointer



```cpp
#include<iostream>
using namespace std;
class Test
{
    int x;
    float y;
public:
    void set()
    {
        x = 20;
        y = 40.3;
    }
    void print()
    {
        cout<<"\n\nx = "<<x<<endl;
        cout<<"\nY = "<<y<<endl;
    }
};
int main()
{
    cout<<"********* Implicit this poniter *********";
    Test obj;
    obj.set();
    obj.print();
    return 0;
}
```

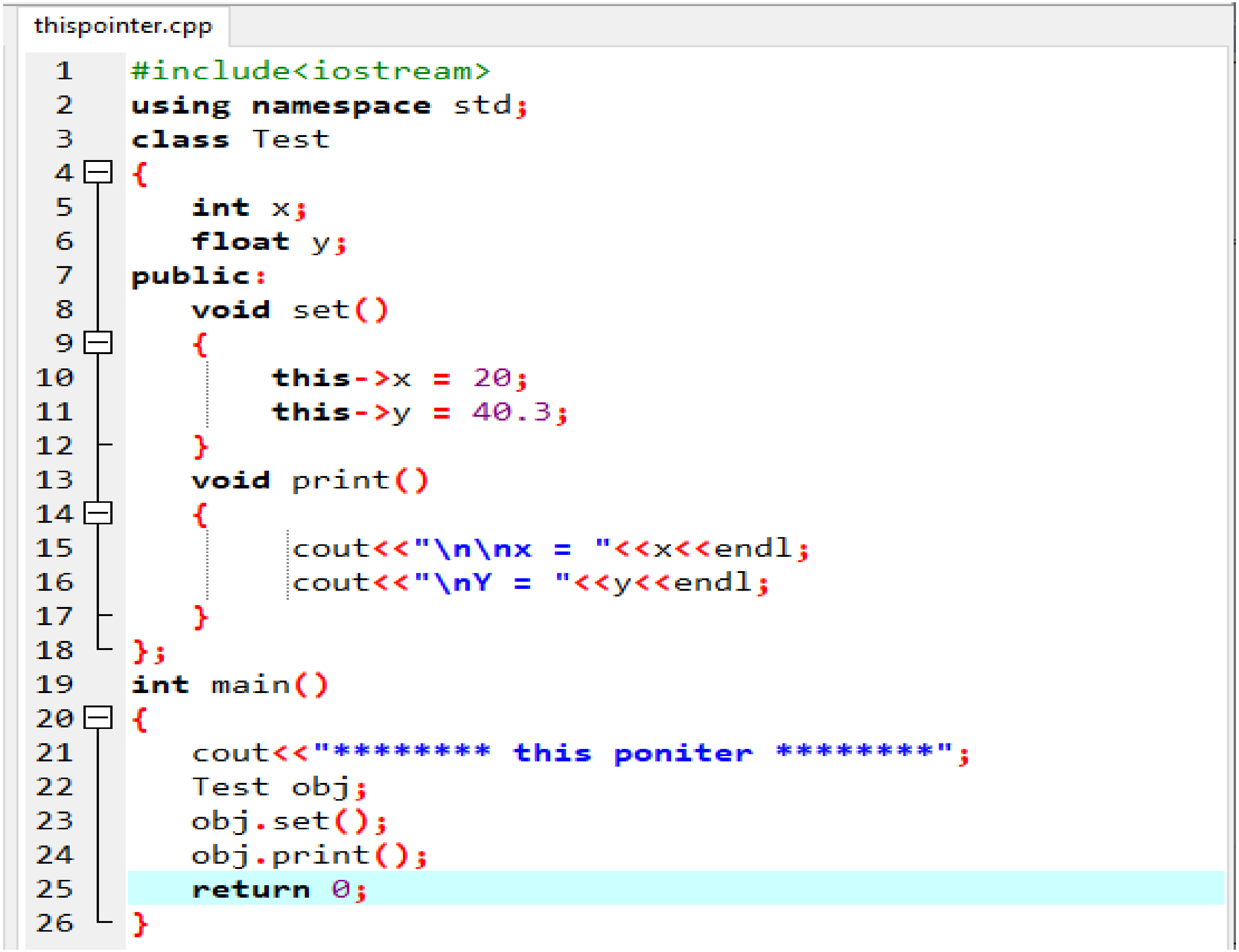
- Output:



Select F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\implicitthispointer.exe

```
******** Implicit this poniter ********

x = 20

Y = 40.3

--------------------------------
Process exited after 0.1492 seconds with return value 0
Press any key to continue . . .
```

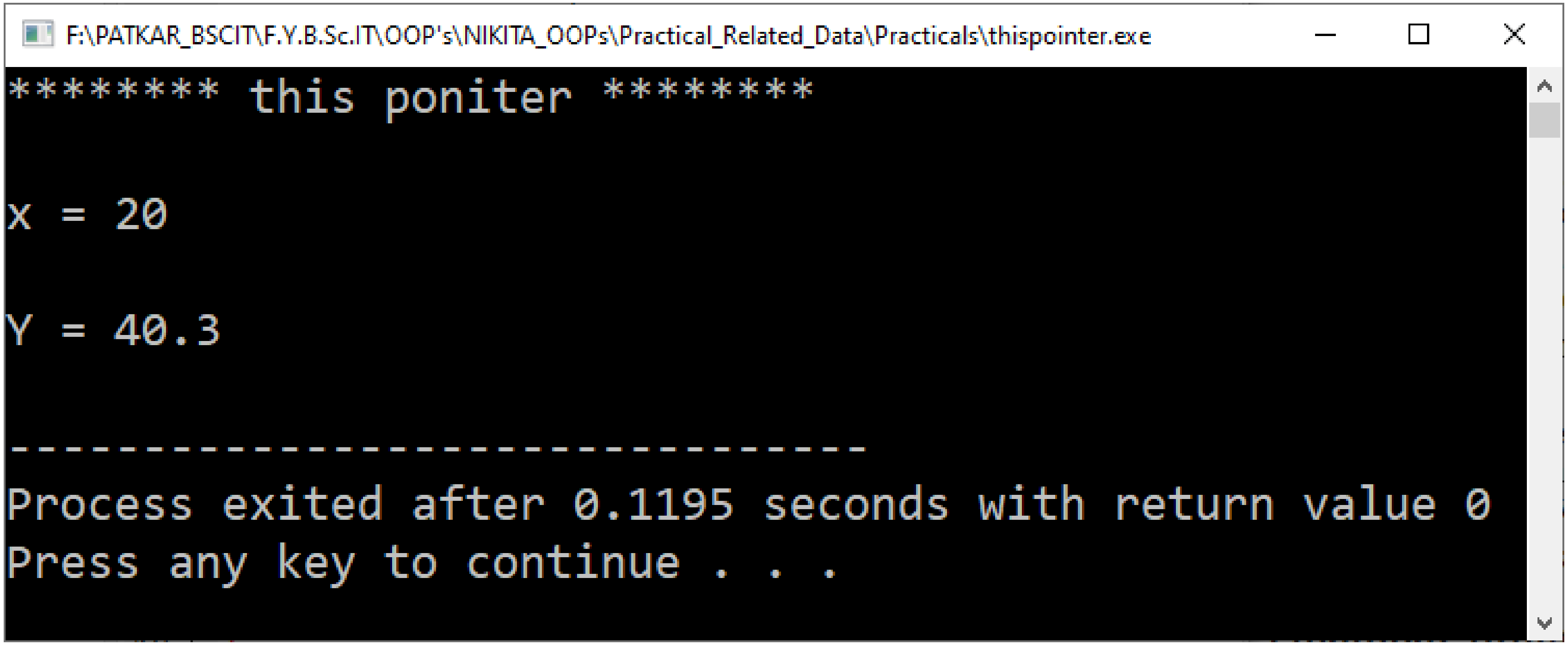
- E.g. Using this pointer



thispointer.cpp

```
#include<iostream>
using namespace std;
class Test
{
    int x;
    float y;
public:
    void set()
    {
        this->x = 20;
        this->y = 40.3;
    }
    void print()
    {
          cout<<"\n\nx = "<<x<<endl;
          cout<<"\nY = "<<y<<endl;
    }
};
int main()
{
    cout<<"********* this poniter *********";
    Test obj;
    obj.set();
    obj.print();
    return 0;
}
```

- Output :

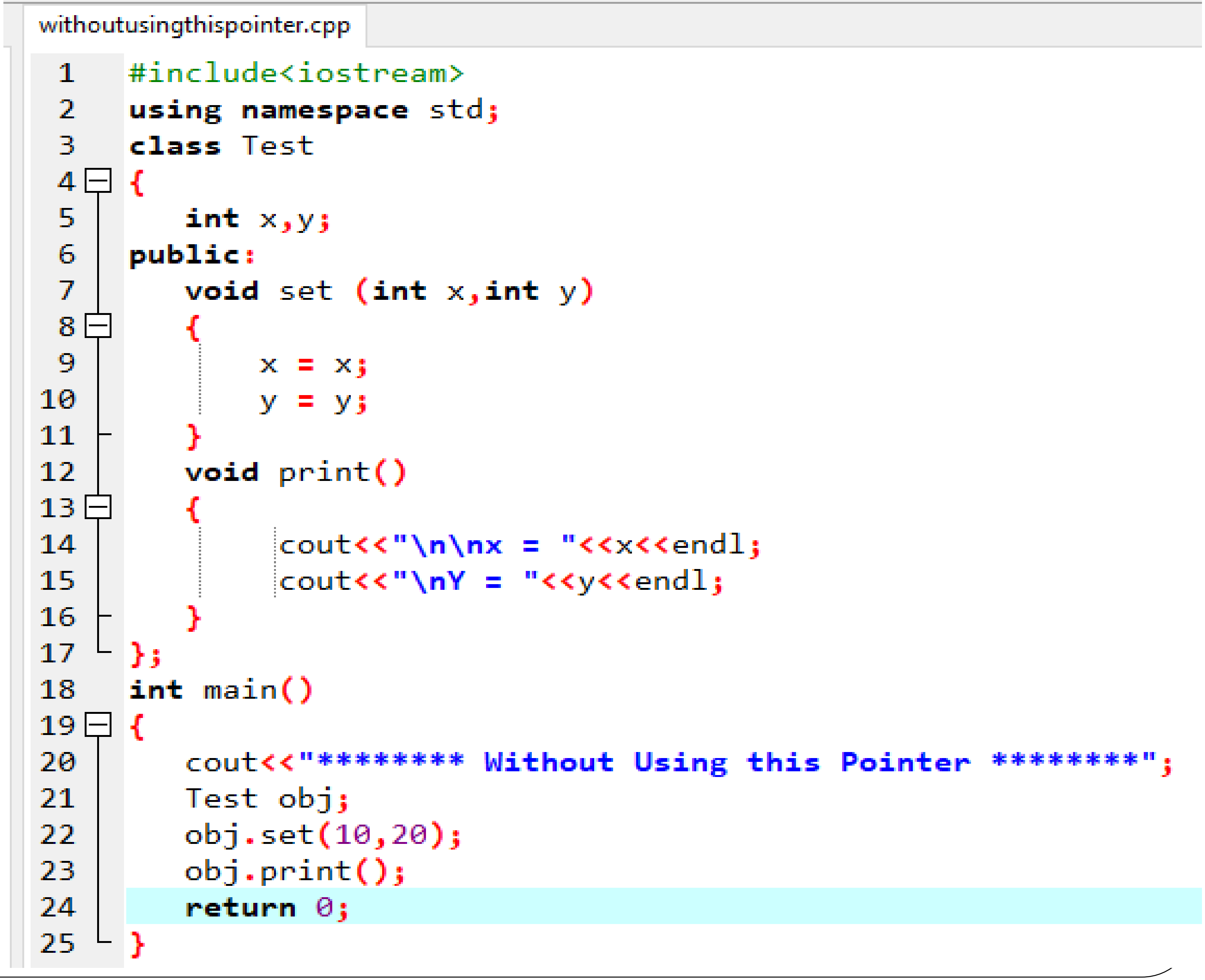


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\thispointer.exe

```
******** this poniter ********

x = 20

Y = 40.3

--------------------------------
Process exited after 0.1195 seconds with return value 0
Press any key to continue . . .
```

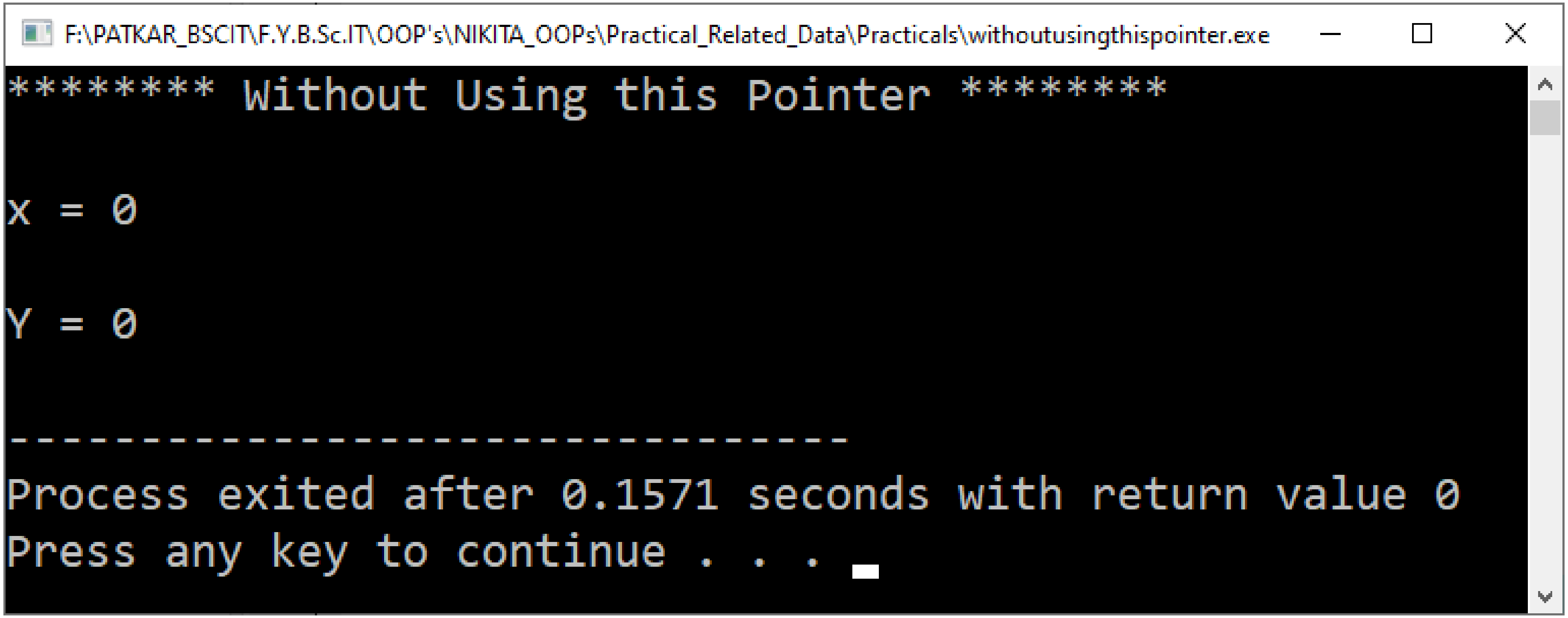
- this pointer is use when local variable is same as member name

Before Using this pointer



```cpp
#include<iostream>
using namespace std;
class Test
{
    int x,y;
public:
    void set (int x,int y)
    {
        x = x;
        y = y;
    }
    void print()
    {
         cout<<"\n\nx = "<<x<<endl;
         cout<<"\nY = "<<y<<endl;
    }
};
int main()
{
    cout<<"******** Without Using this Pointer ********";
    Test obj;
    obj.set(10,20);
    obj.print();
    return 0;
}
```

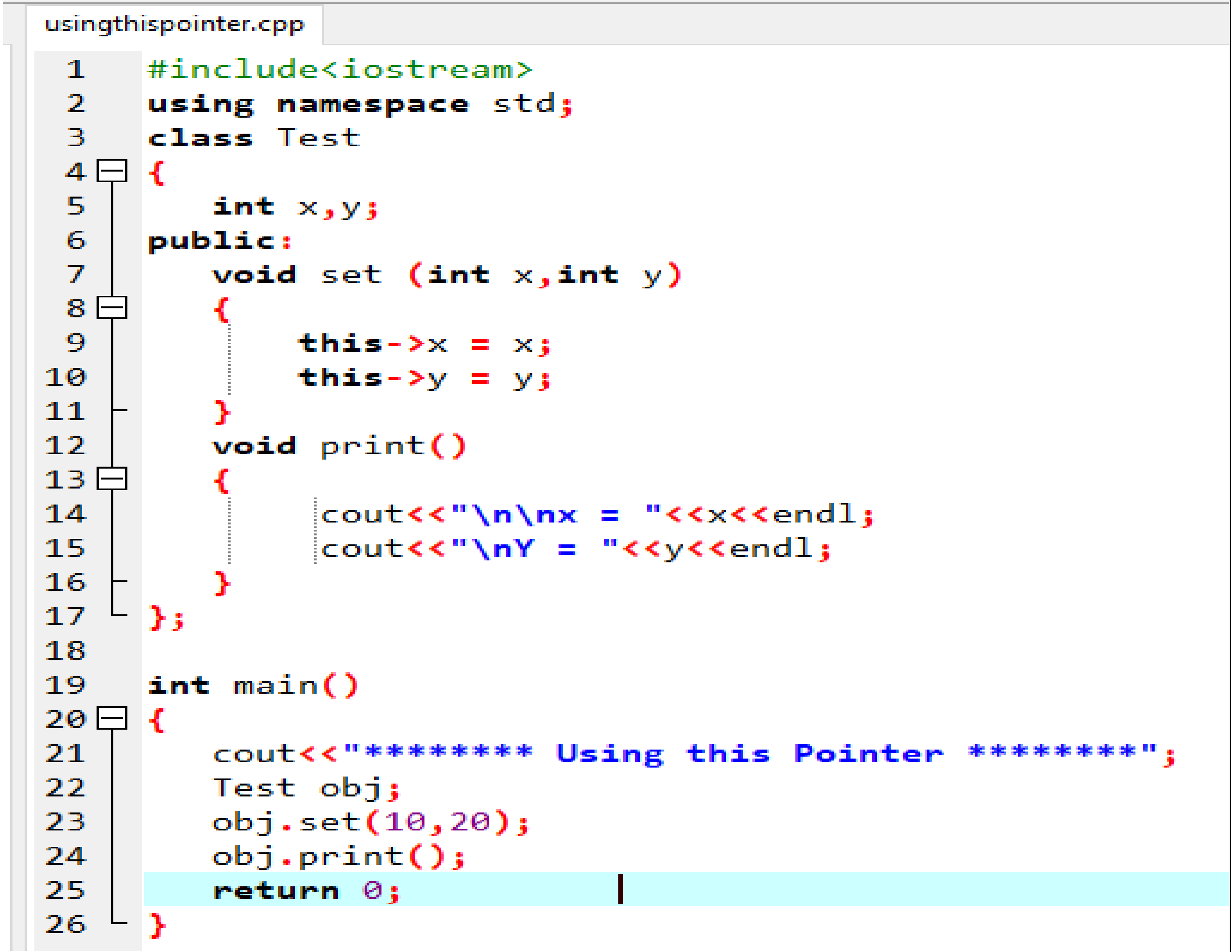
- Output:



F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\withoutusingthispointer.exe

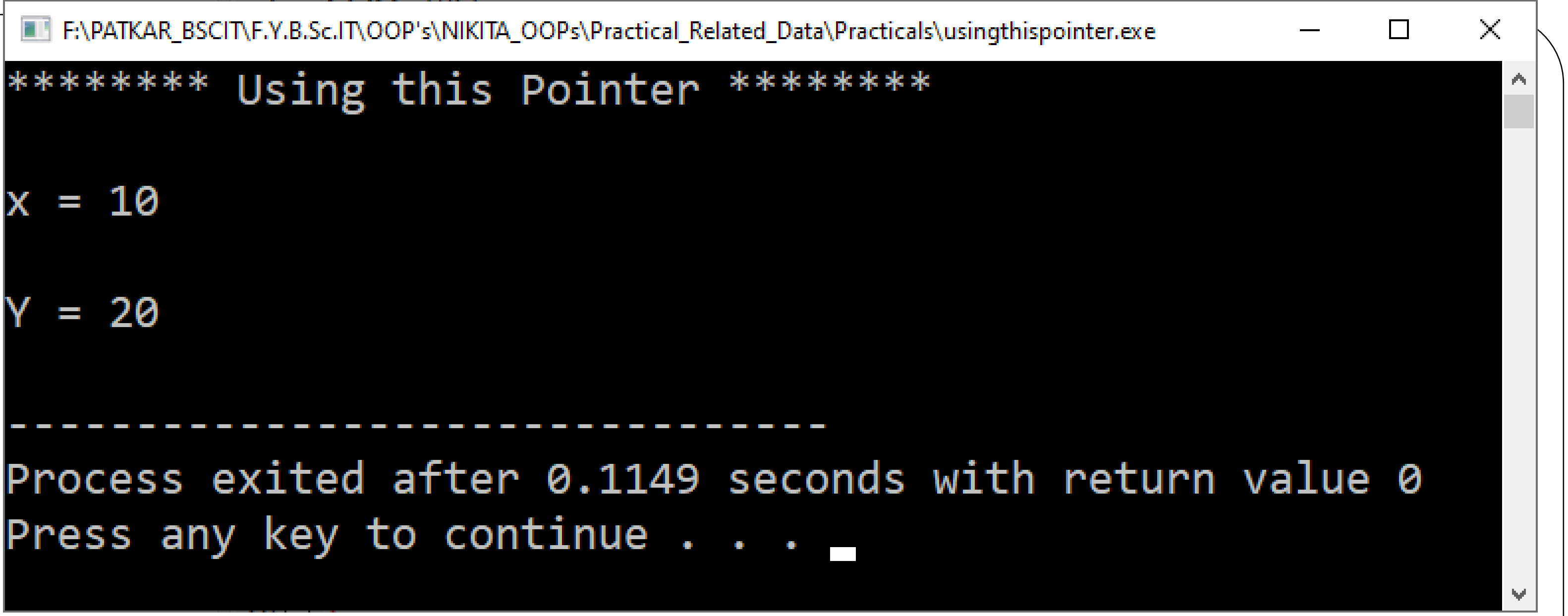
```
******** Without Using this Pointer ********

X = 0

Y = 0

--------------------------------
Process exited after 0.1571 seconds with return value 0
Press any key to continue . . .
```

After Using this pointer



```cpp
#include<iostream>
using namespace std;
class Test
{
    int x,y;
public:
    void set (int x,int y)
    {
        this->x = x;
        this->y = y;
    }
    void print()
    {
        cout<<"\n\nx = "<<x<<endl;
        cout<<"\nY = "<<y<<endl;
    }
};

int main()
{
    cout<<"********* Using this Pointer *********";
    Test obj;
    obj.set(10,20);
    obj.print();
    return 0;
}
```

- Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\usingthispointer.exe

```
******** Using this Pointer ********

x = 10

Y = 20

--------------------------------
Process exited after 0.1149 seconds with return value 0
Press any key to continue . . .
```

- Here you can see that we have **two data members x and y**.
- In **member function set()** we have **two local variables** (parameter in function) **having same name** as **data members name**(variable).
- In such case if you want to **assign** the **local variable**(parameter in function) **value to** the **data members**(variable) **then** you **won't be able** to do **until unless** you **use** **this pointer**, **because** the compiler won't know that you are referring to object's data members unless you use this pointer

What is Binding in C++?

- Binding is the process of **linking the function call with the place where the function definition** is actually written.
- So that when a function call is made, it can be ascertained(make sure) where the control has to be transferred.
- Binding is also termed as linking.
- Binding is of two types :

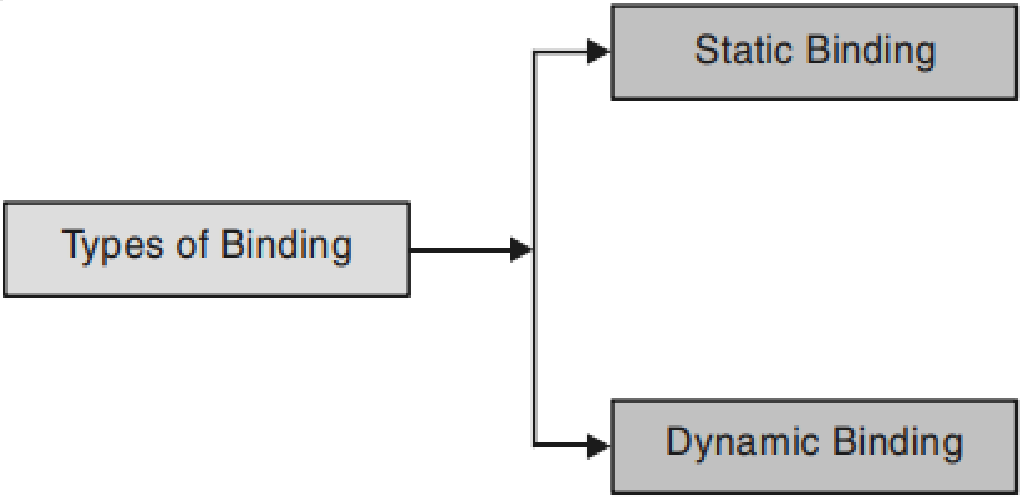


Figure 10.1. *Types of binding in C++*

Static Binding

- When it is **known** at **compile time which function will be called in response to a function call**, binding is known as **static binding**, compile time or early binding.
- Static binding is called so before program executes it is fixed that a particular function be called in response to a function call.
- Each time program executes same function will be called.
- As the linking is done early to the execution of the program executes same function will be called. As the linking is done at compile time it is known as compile time binding.

Dynamic Binding

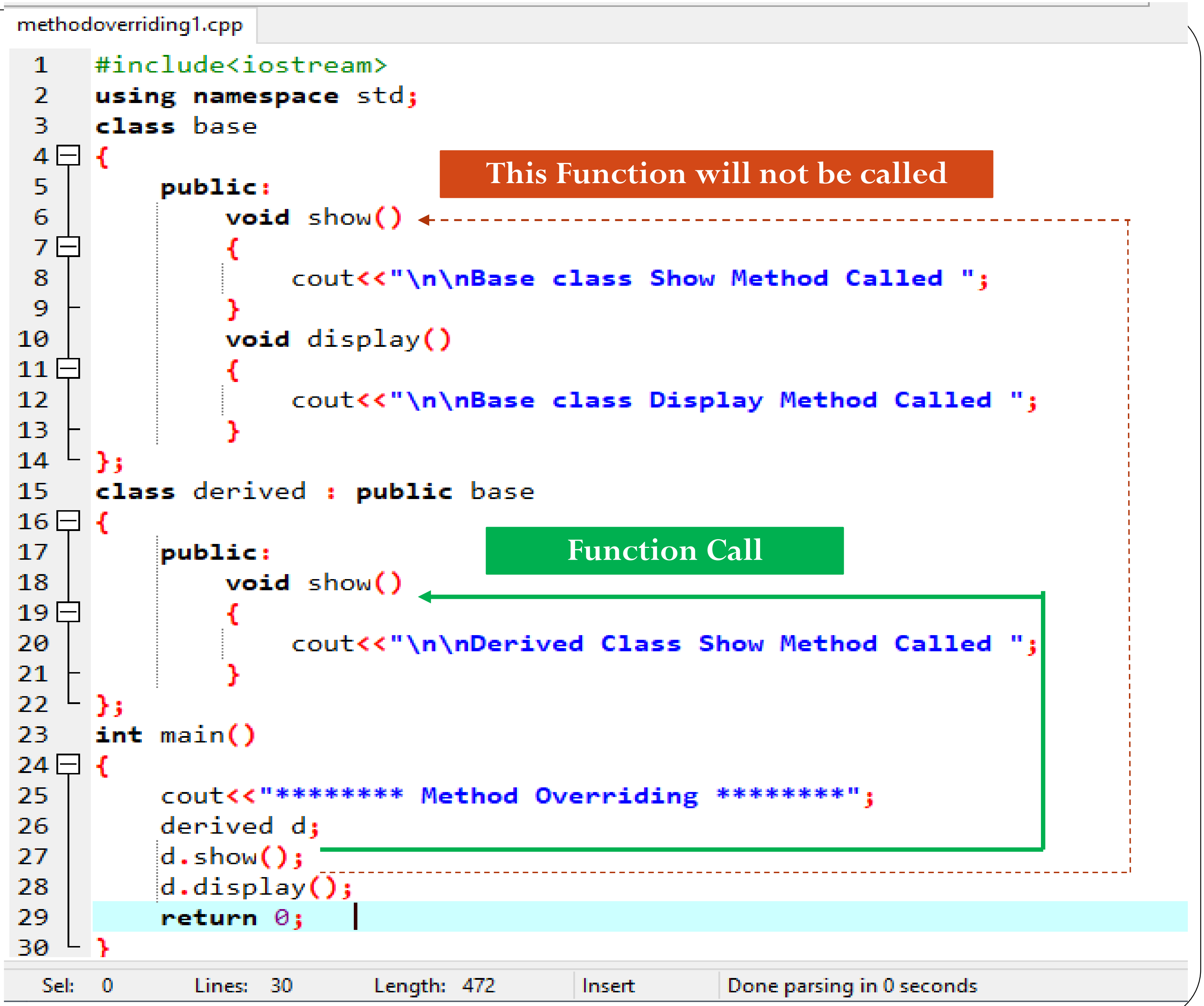
- When it is not certain that **which function is called in response to a function call**, binding is delayed till program executes.
- At **run time the decision is taken** as to **which function is called** in response to a function call.
- This type of binding is known as late binding, runtime binding or **dynamic binding**.
- Dynamic binding is **based purely on finding the address of pointers** and as addresses are generated during run time or when time run or when program executes, this type of binding is known as run-time or execution time binding.

Method Overriding (Function Overriding)

- When we redefine a **base class's function in** the **derived class** with the **same function signature**(name) but with a **different implementation**, it is called **Method or Function overriding**
- **Both** the **base class** and **derived class** have a **member function** with **same name** and **arguments** (number and type of arguments) and if an **object** has been **created** of **derived class** and **call** the member **function** which exist in **both the classes** (base and derived), the member **function of** the **derived class** is **invoked** and the **function** of the **base class** get **ignored**. This feature in c++ is known as **Method Overriding**.

- In function overriding, the function in parent class is called the overridden function and function in child class is called overriding function.
- The function in derived class overrides the function in base class

E.g.

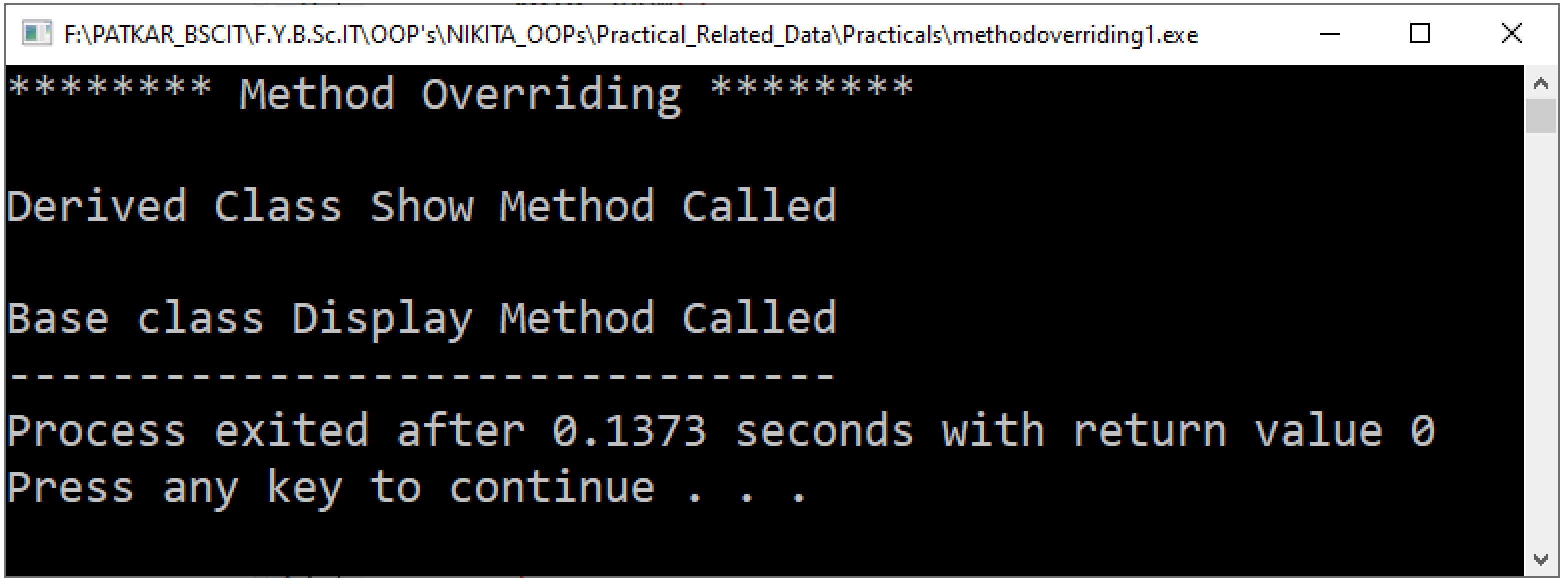


```cpp
#include<iostream>
using namespace std;
class base
{
    public:
        void show()
        {
            cout<<"\n\nBase class Show Method Called ";
        }
        void display()
        {
            cout<<"\n\nBase class Display Method Called ";
        }
};
class derived : public base
{
    public:
        void show()
        {
            cout<<"\n\nDerived Class Show Method Called ";
        }
};
int main()
{
    cout<<"******** Method Overriding ********";
    derived d;
    d.show();
    d.display();
    return 0;
}
```

This Function will not be called

Function Call

Output :


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\methodoverriding1.exe

```
********* Method Overriding *********

Derived Class Show Method Called

Base class Display Method Called
--------------------------------
Process exited after 0.1373 seconds with return value 0
Press any key to continue . . .
```

How to call overridden function from the child class

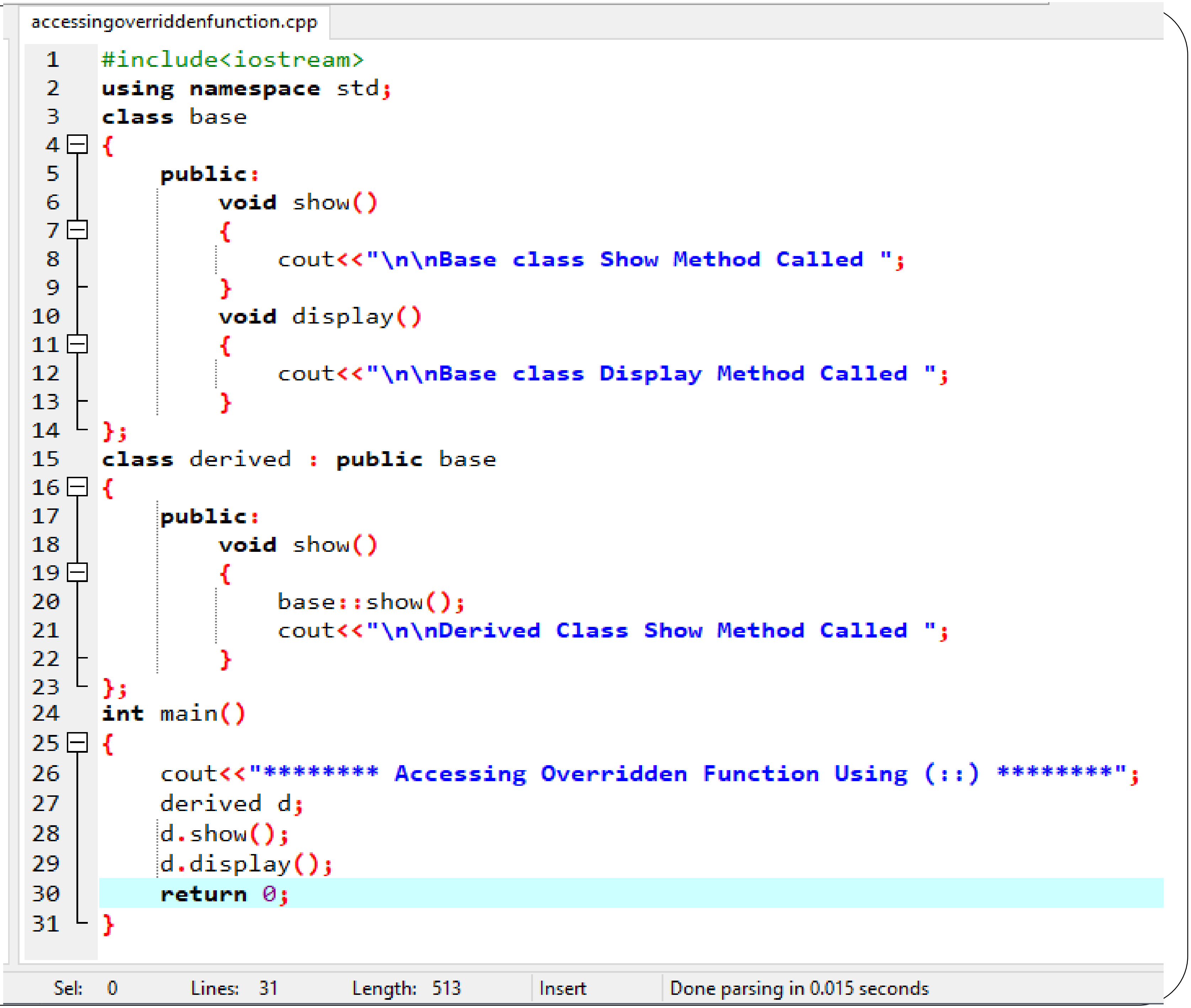
- If you want to call the Overridden function from overriding function then you can do it like this:
- The overridden function of the base class will be accessed by the derived class using the scope resolution operator (::)
- Syntax

base_class_name::function_name

- e.g.

base::show();

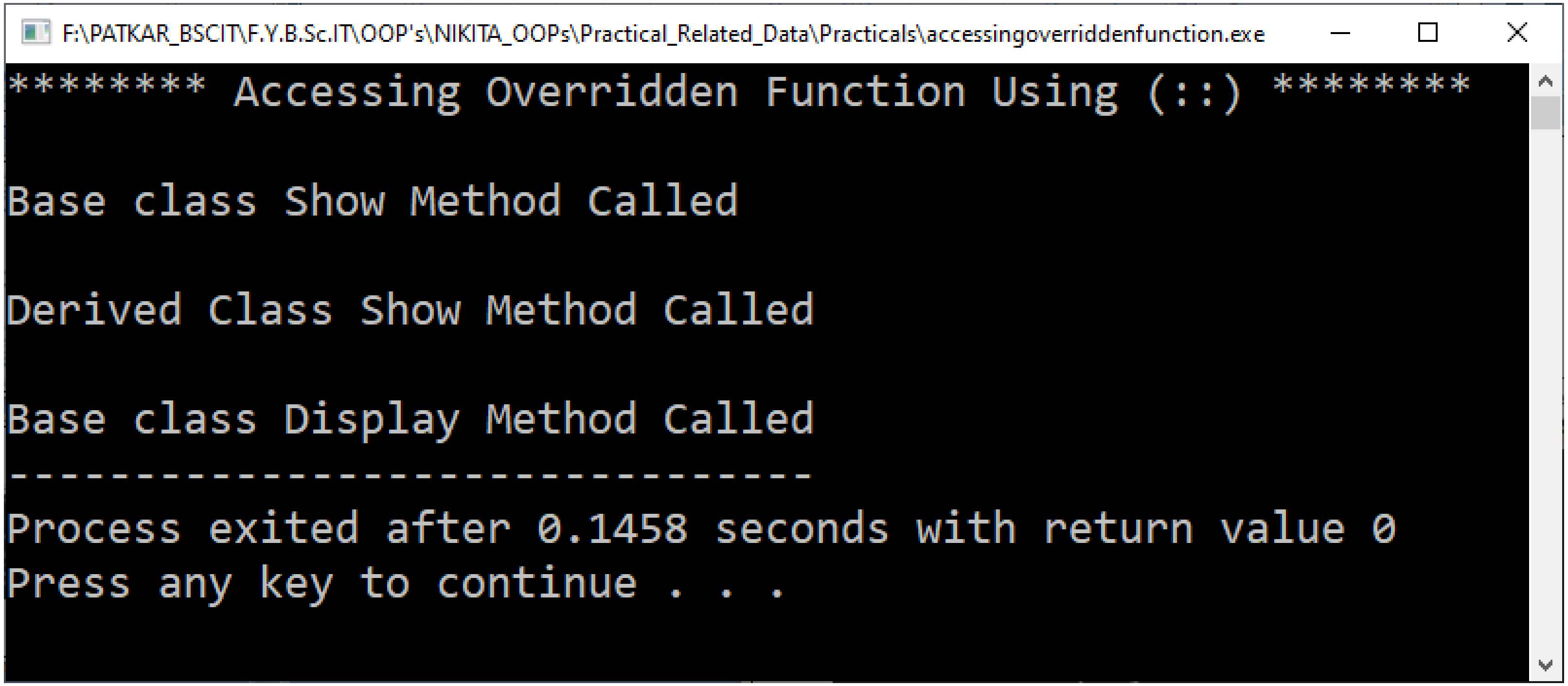
E.g.



```
#include<iostream>
using namespace std;
class base
{
    public:
        void show()
        {
            cout<<"\n\nBase class Show Method Called ";
        }
        void display()
        {
            cout<<"\n\nBase class Display Method Called ";
        }
};
class derived : public base
{
    public:
        void show()
        {
            base::show();
            cout<<"\n\nDerived Class Show Method Called ";
        }
};
int main()
{
    cout<<"******** Accessing Overridden Function Using (::) ********";
    derived d;
    d.show();
    d.display();
    return 0;
}
```

Sel: 0 Lines: 31 Length: 513 Insert Done parsing in 0.015 seconds

Output :



F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\accessingoverriddenfunction.exe

```
******** Accessing Overridden Function Using (::) ********

Base class Show Method Called

Derived Class Show Method Called

Base class Display Method Called
--------------------------------
Process exited after 0.1458 seconds with return value 0
Press any key to continue . . .
```

Virtual Function

- **Virtual function** is a function that is **declared** as **virtual** in a **base class** and **redefined** in the **derived class**.
- When there is **same function name** in **both** the **base** **and** **derived** **classes**, the **function** in **base class** is **declared** as **virtual**.
- To **create** **virtual function**, precede the **function's declaration** in the **base class** with the **keyword** **virtual**
- When a function is made virtual, C++ determines **which function** to **use** at run time based on the **type of object** **pointed** **to** **by** the **base pointer**, **rather than** the **type of the pointer**.

Why we use Virtual Function (Use of Virtual Function)

- **Base class pointer** can **point** to **derived class object**. In this case, **using base class pointer** if we **call** some **function which** is **in both classes**, **then base class function** is **invoked**.
- But if we **want** to **invoke derived class function** using **base class pointer**, it can be **achieved** by **defining the function** as **virtual** in **base class**, this is how virtual functions support runtime polymorphism. (So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the '**virtual**' function)

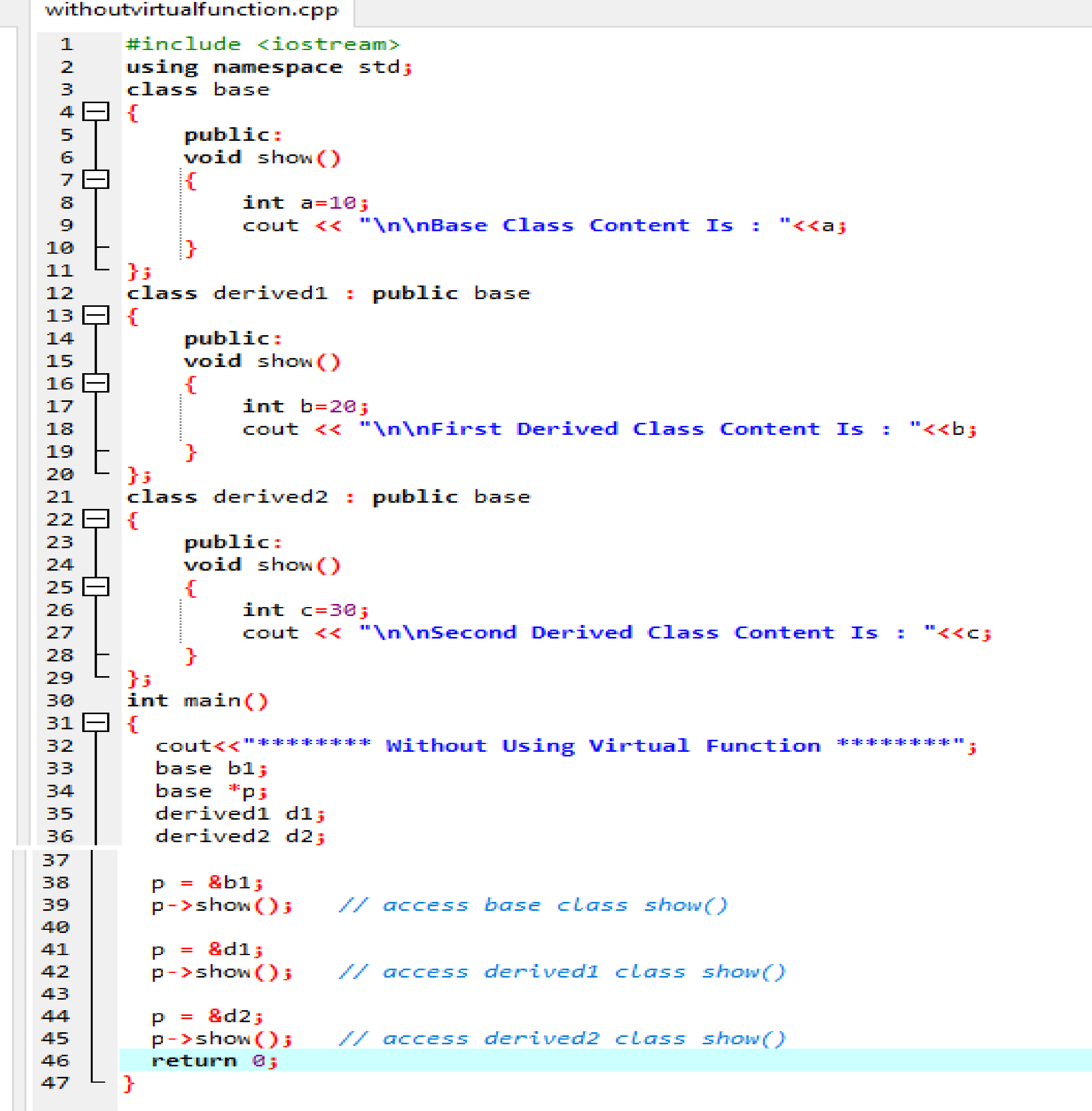
- If the **function** is made **virtual**, then the **compiler** will **determine which function** is to **execute** at the run time **on** the **basis** of the **assigned address** **to the pointer** **of** the **base class**

Rules of Virtual Function

- Virtual functions **must** be **members of some class**.
- They are **accessed** through **object pointers**.
- Virtual functions **cannot** be **static members**.
- We **cannot** have a **virtual constructor**, but we **can have** a **virtual destructor**
- They **can** be a **friend** of **another class**.
- A virtual function **must** be **defined** in the **base class**, even **though it is not used**.
- The **prototypes** (parameters in function) of a **virtual function of** the **base class** and all the **derived classes must** be **identical** (same). If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions

e.g.

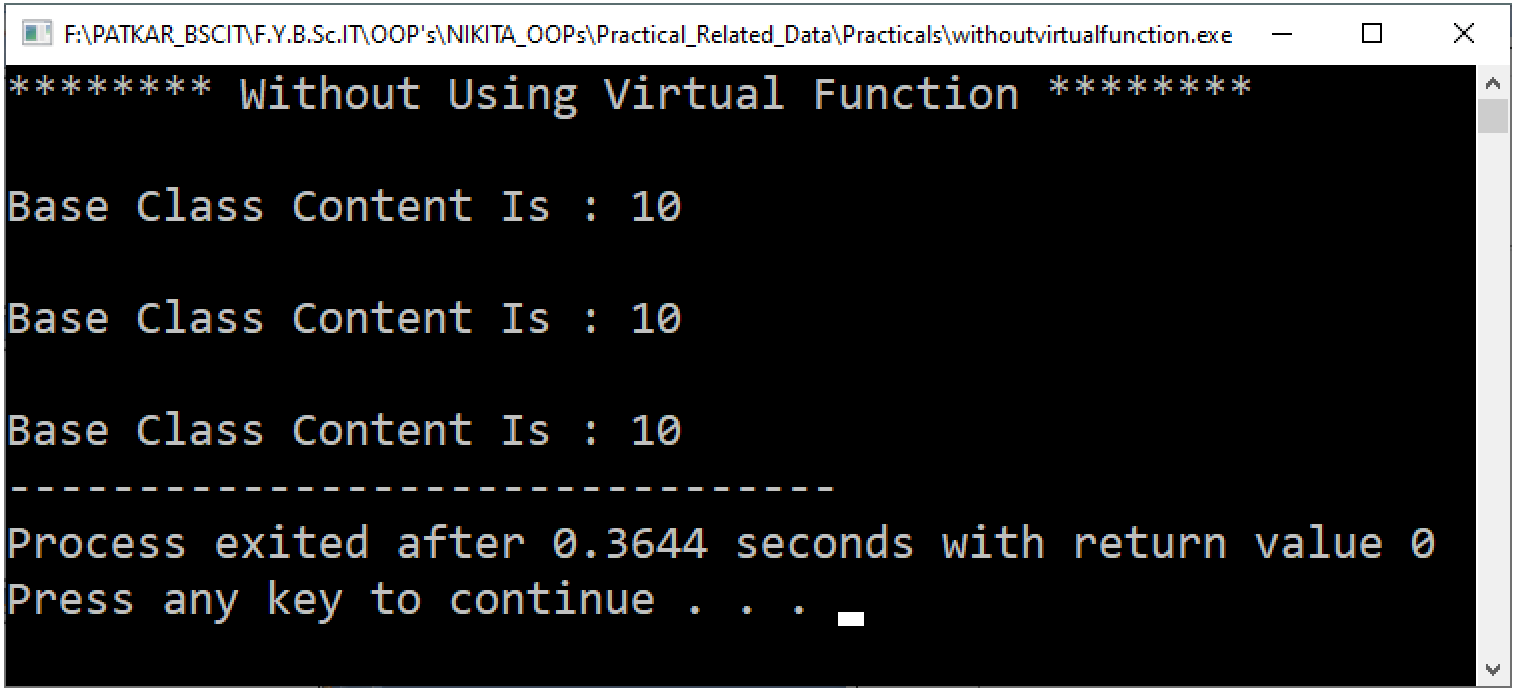
Without Virtual Function



withoutvirtualfunction.cpp

```cpp
#include <iostream>
using namespace std;
class base
{
      public:
      void show()
      {
            int a=10;
            cout << "\n\nBase Class Content Is : "<<a;
      }
};
class derived1 : public base
{
      public:
      void show()
      {
            int b=20;
            cout << "\n\nFirst Derived Class Content Is : "<<b;
      }
};
class derived2 : public base
{
      public:
      void show()
      {
            int c=30;
            cout << "\n\nSecond Derived Class Content Is : "<<c;
      }
};
int main()
{
   cout<<"******** Without Using Virtual Function ********";
   base b1;
   base *p;
   derived1 d1;
   derived2 d2;

   p = &b1;
   p->show();    // access base class show()

   p = &d1;
   p->show();    // access derived1 class show()

   p = &d2;
   p->show();    // access derived2 class show()
   return 0;
}
```

Output :

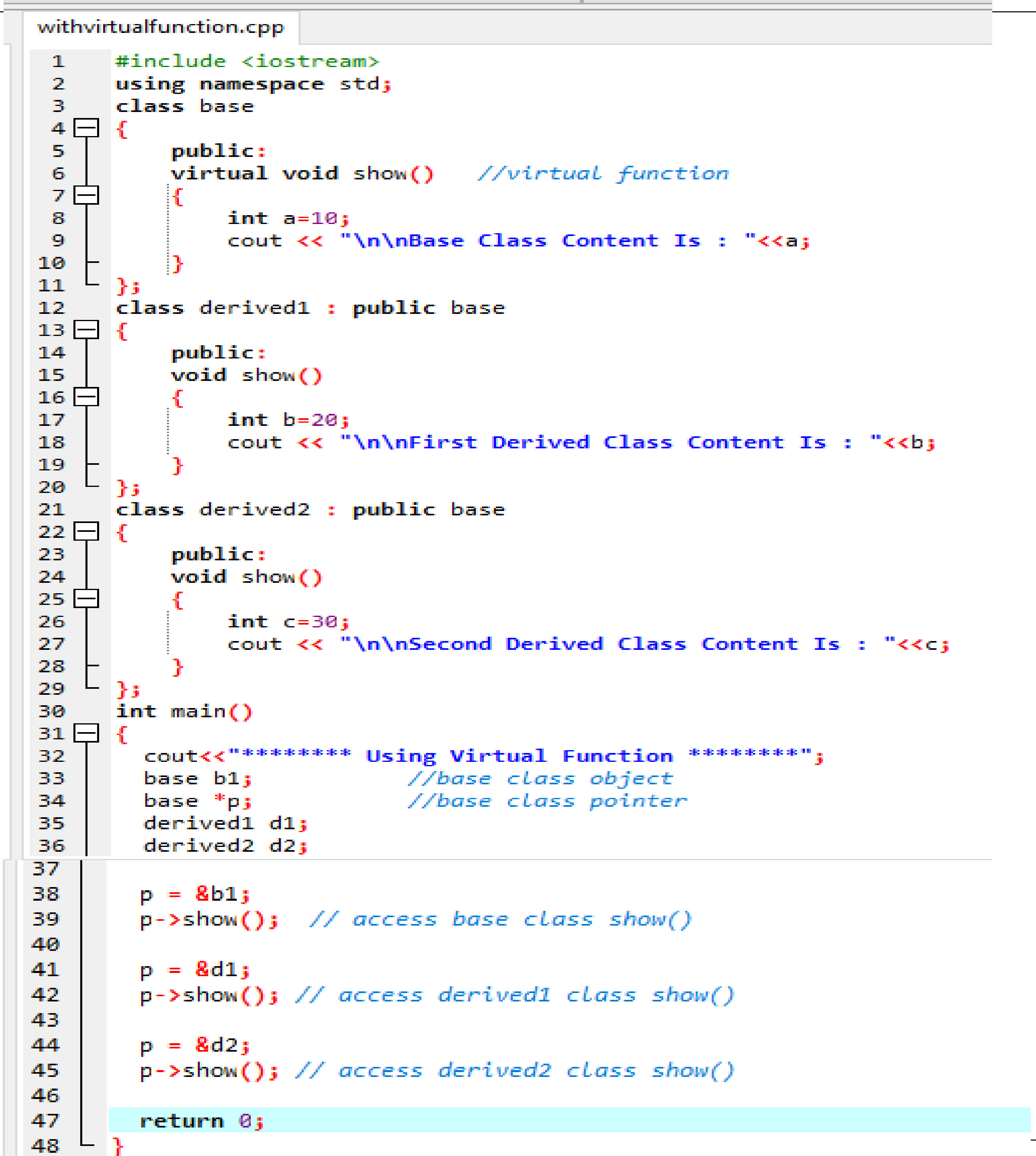


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\withoutvirtualfunction.exe

```
********* Without Using Virtual Function *********

Base Class Content Is : 10

Base Class Content Is : 10

Base Class Content Is : 10
--------------------------------
Process exited after 0.3644 seconds with return value 0
Press any key to continue . . .
```

e.g.

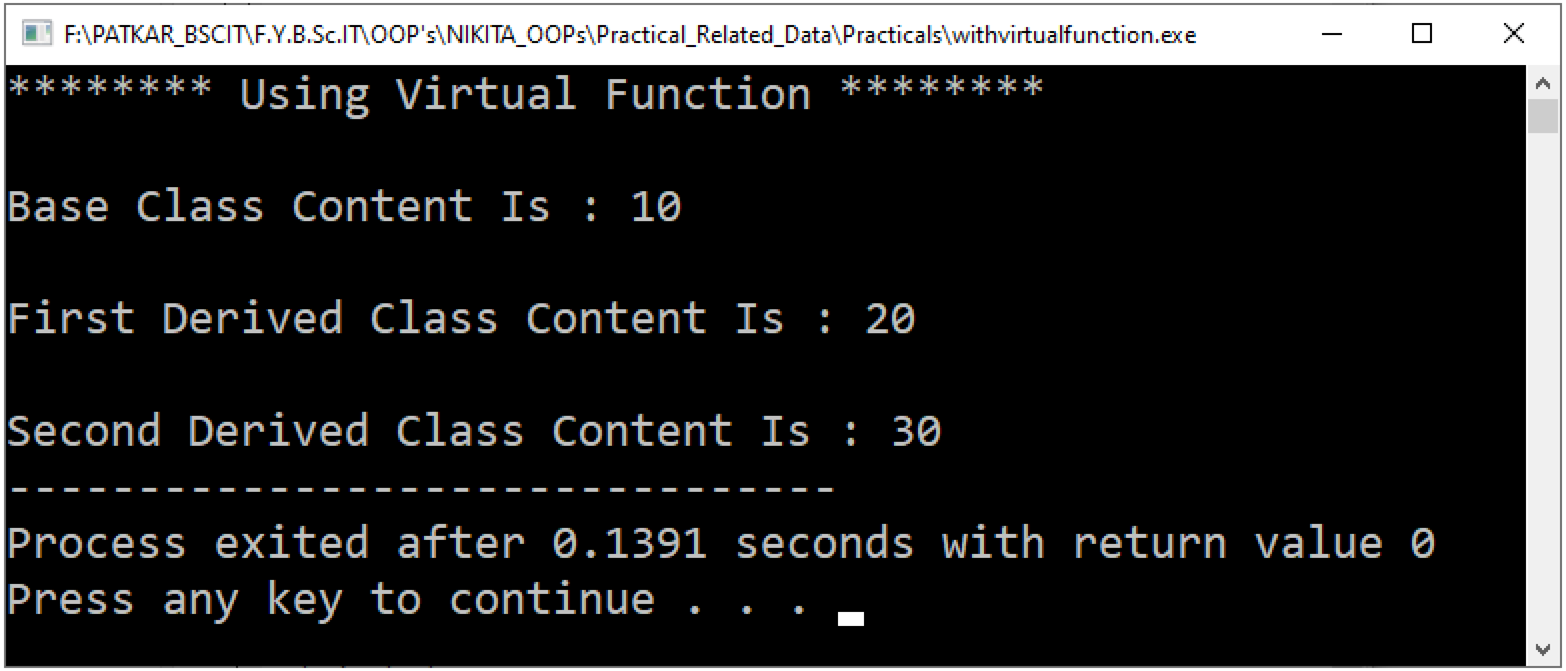
Using Virtual Function



withvirtualfunction.cpp

```cpp
#include <iostream>
using namespace std;
class base
{
    public:
    virtual void show()   //virtual function
    {
        int a=10;
        cout << "\n\nBase Class Content Is : "<<a;
    }
};
class derived1 : public base
{
    public:
    void show()
    {
        int b=20;
        cout << "\n\nFirst Derived Class Content Is : "<<b;
    }
};
class derived2 : public base
{
    public:
    void show()
    {
        int c=30;
        cout << "\n\nSecond Derived Class Content Is : "<<c;
    }
};
int main()
{
  cout<<"******** Using Virtual Function ********";
  base b1;               //base class object
  base *p;               //base class pointer
  derived1 d1;
  derived2 d2;

  p = &b1;
  p->show();  // access base class show()

  p = &d1;
  p->show(); // access derived1 class show()

  p = &d2;
  p->show(); // access derived2 class show()

  return 0;
}
```

Output :



F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\withvirtualfunction.exe

```
********* Using Virtual Function *********

Base Class Content Is : 10

First Derived Class Content Is : 20

Second Derived Class Content Is : 30
--------------------------------
Process exited after 0.1391 seconds with return value 0
Press any key to continue . . .
```

Pure Virtual Function and Abstract Class

Pure Virtual Function

- A virtual function which is **not used** for performing any task.
- When the **function** has **no definition**, such function is **known** as "**do-nothing**" **function**.
- The "**do-nothing**" **function** is **known** as a **pure virtual function**. A **pure virtual function** is a **function declared in** the **base class** that **has no definition** relative to the base class (no definition in base class).
- A **class containing** the **pure virtual function cannot** be **used** to **declare** the **objects** of its **own**, such **classes** are **known** as **abstract base classes**.
- Pure virtual function can be defined as:
- Syntax

```
virtual function_name() = 0;
```

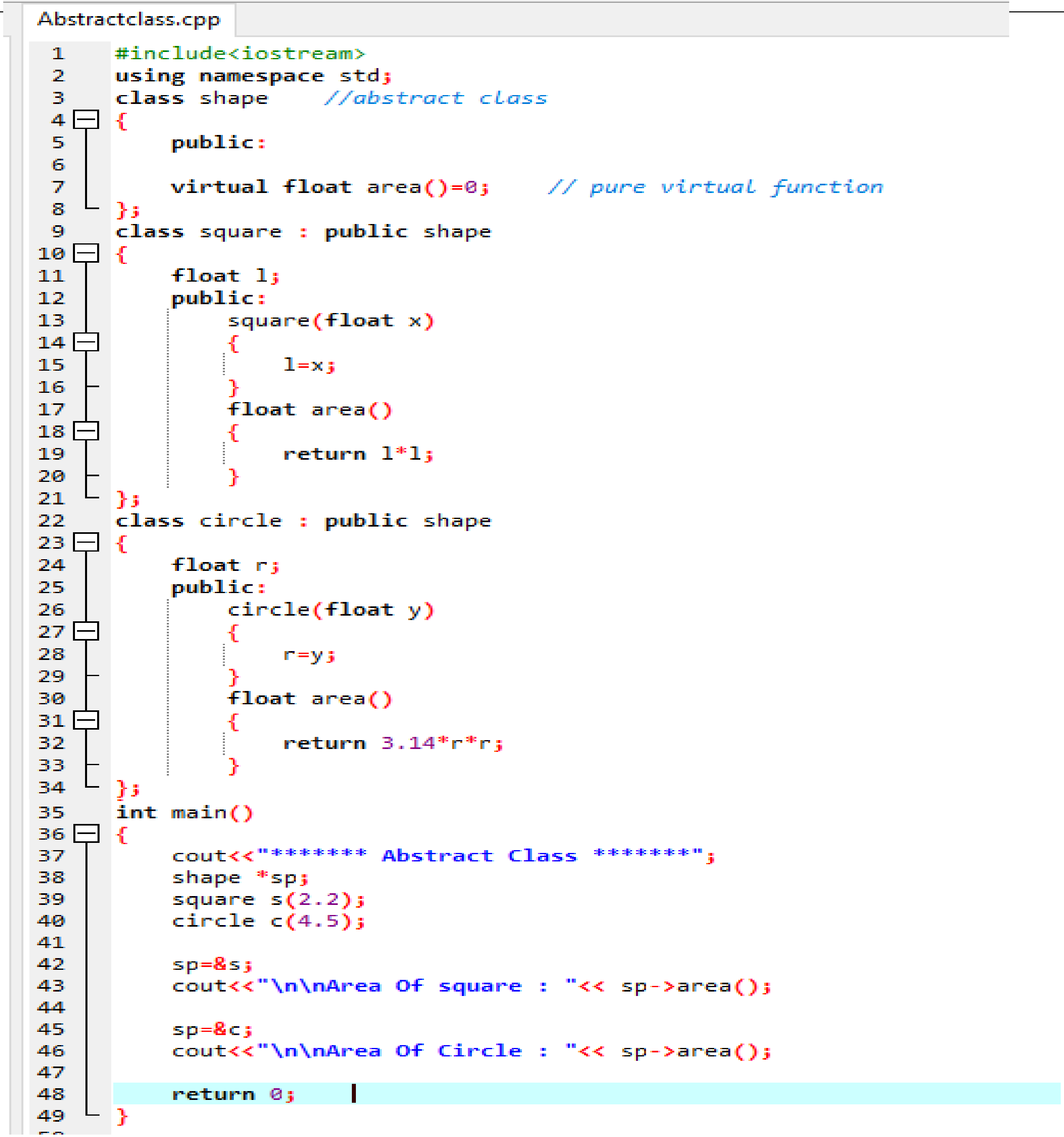
- e.g.

```
virtual void display() = 0;
```

Abstract Class

- An **abstract class** is one that is **not** used to **create objects**
- Though objects of an abstract class cannot be created, however, one can use pointers and references to abstract class types
- An **abstract class** is designed only to act as a **base class** (to be inherited by other classes).
- A **class** should **contain at least one pure virtual function** to be called as **abstract**
- **Declaration of a Abstract Class:**
  - If expression **= 0** is added to a **virtual function**, then that function becomes **pure virtual function**
  - **Note :** that adding =0 to virtual function does not assign value, it simply indicates the virtual function is a pure function.

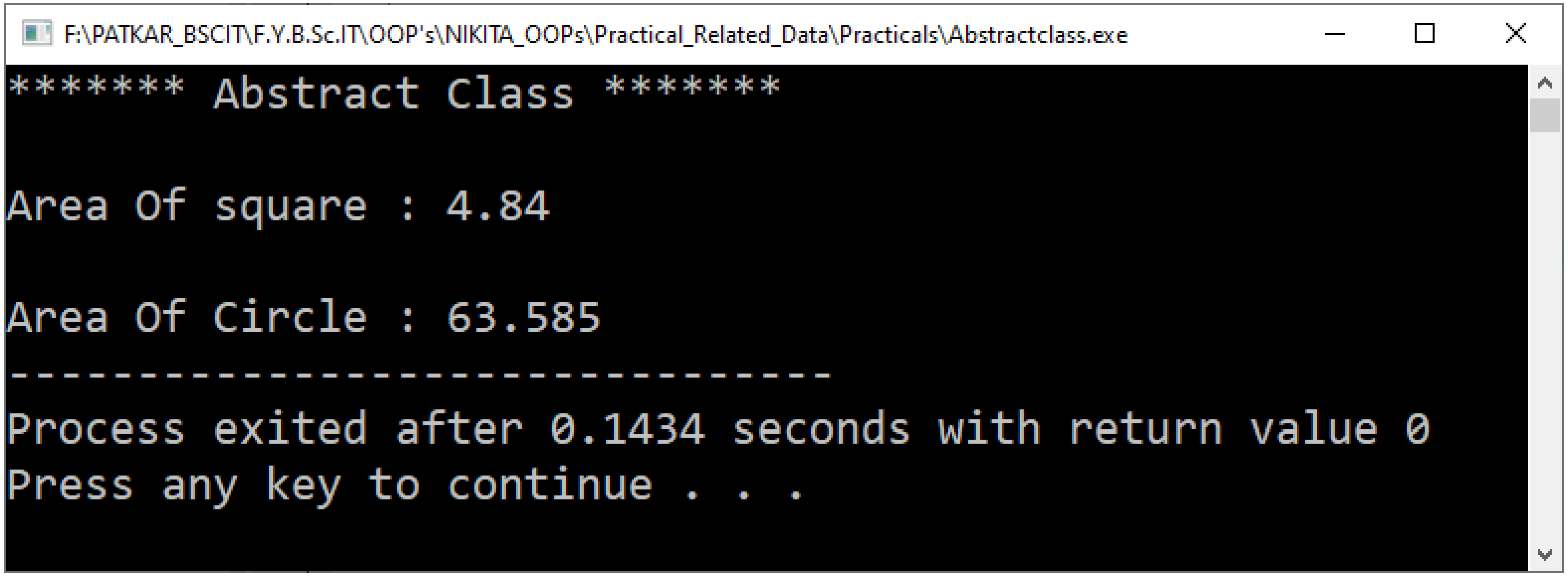
e.g.



Abstractclass.cpp

```
#include<iostream>
using namespace std;
class shape    //abstract class
{
    public:

    virtual float area()=0;    // pure virtual function
};
class square : public shape
{
    float l;
    public:
        square(float x)
        {
            l=x;
        }
        float area()
        {
            return l*l;
        }
};
class circle : public shape
{
    float r;
    public:
        circle(float y)
        {
            r=y;
        }
        float area()
        {
            return 3.14*r*r;
        }
};
int main()
{
    cout<<"******* Abstract Class *******";
    shape *sp;
    square s(2.2);
    circle c(4.5);

    sp=&s;
    cout<<"\n\nArea Of square : "<< sp->area();

    sp=&c;
    cout<<"\n\nArea Of Circle : "<< sp->area();

    return 0;
}
```

Output :



F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\Abstractclass.exe

```
******* Abstract Class *******

Area Of square : 4.84

Area Of Circle : 63.585
--------------------------------
Process exited after 0.1434 seconds with return value 0
Press any key to continue . . .
```

Virtual Destructor

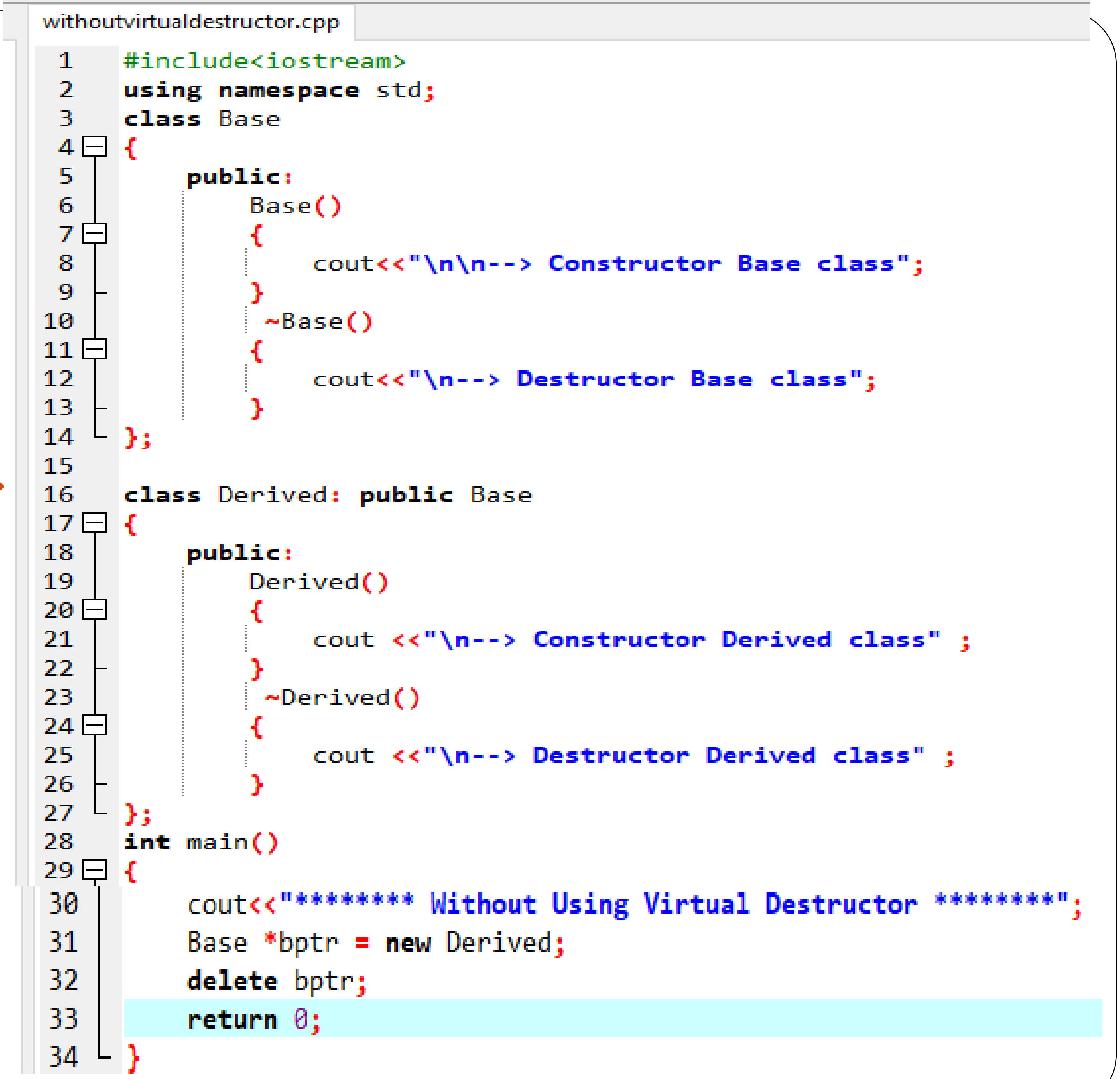
- In C++ a **destructor** is generally used to **deallocate memory** and do some other cleanup for a class object and it's class members whenever an **object is destroyed**.
- Destructors are distinguished by the tilde, the **(~)** that appears in front of the destructor name.
- A **virtual destructor** is used to **free up the memory space allocated <u>by</u>** the **derived class object** or instance while deleting instances of the derived class **using** a **base class pointer** object. (*Deleting a derived class object using a pointer to a base class, the base class should be defined with a virtual destructor.*)
- 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 <u>first</u>** and **then base class <u>to</u> release** the <u>space occupied by both destructors</u>.
- In order to define a **virtual destructor**, the **keyword virtual** is used **before** the **tilde (~)** symbol.

Why we use Virtual Destructor (Use of Virtual Destructor)

- When a **pointer** **object** **of the** base class is **deleted** that **points to the** **derived class**, **only** the **parent class** **destructor** is **called**. In this way, it ***skips*** **calling** the **derived class's** **destructor**
- when we use virtual **keyword preceded by the** destructor **tilde (~) sign inside the** **base class**, it **guarantees** that **first** the **derived class's** destructor **is called**. ***Then*** the **base class's destructor** **is called** to **release** the **space occupied** ***by both*** destructors **in the inheritance class**

e.g.

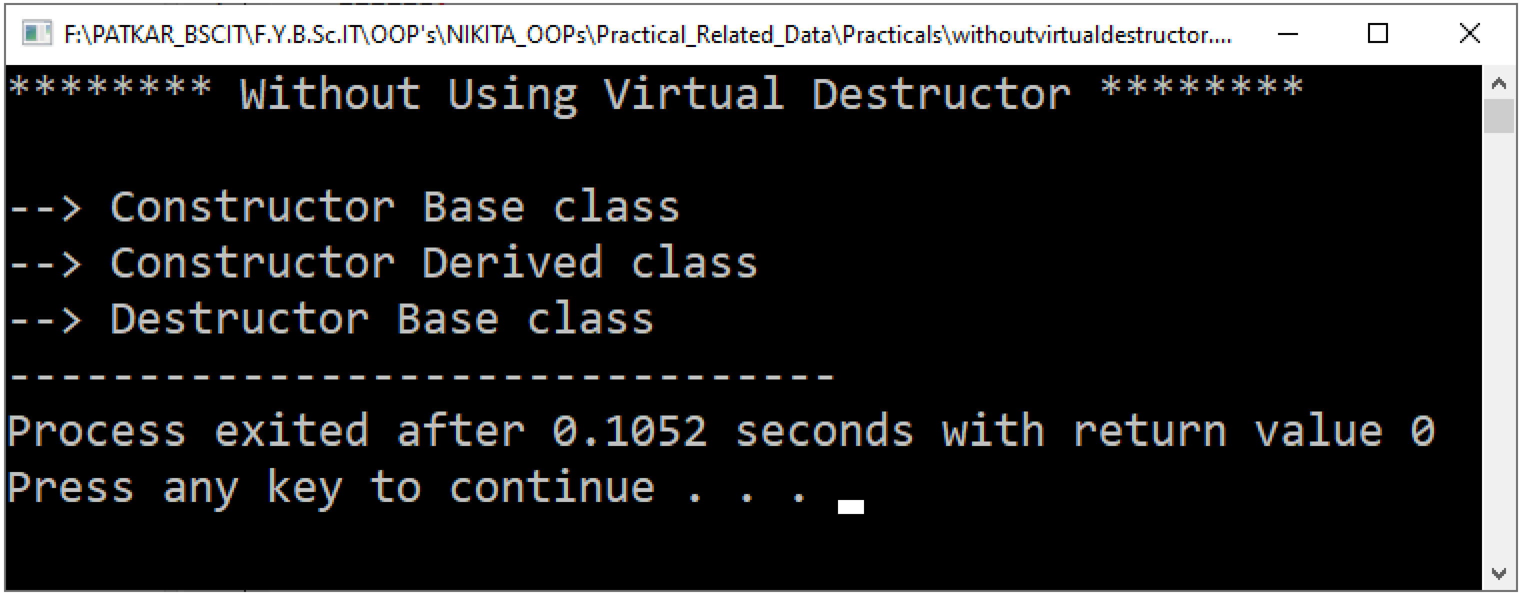
Without Using Virtual Destructor



withoutvirtualdestructor.cpp

```cpp
#include<iostream>
using namespace std;
class Base
{
    public:
        Base()
        {
            cout<<"\n\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()
{
    cout<<"******** Without Using Virtual Destructor ********";
    Base *bptr = new Derived;
    delete bptr;
    return 0;
}
```

Output :



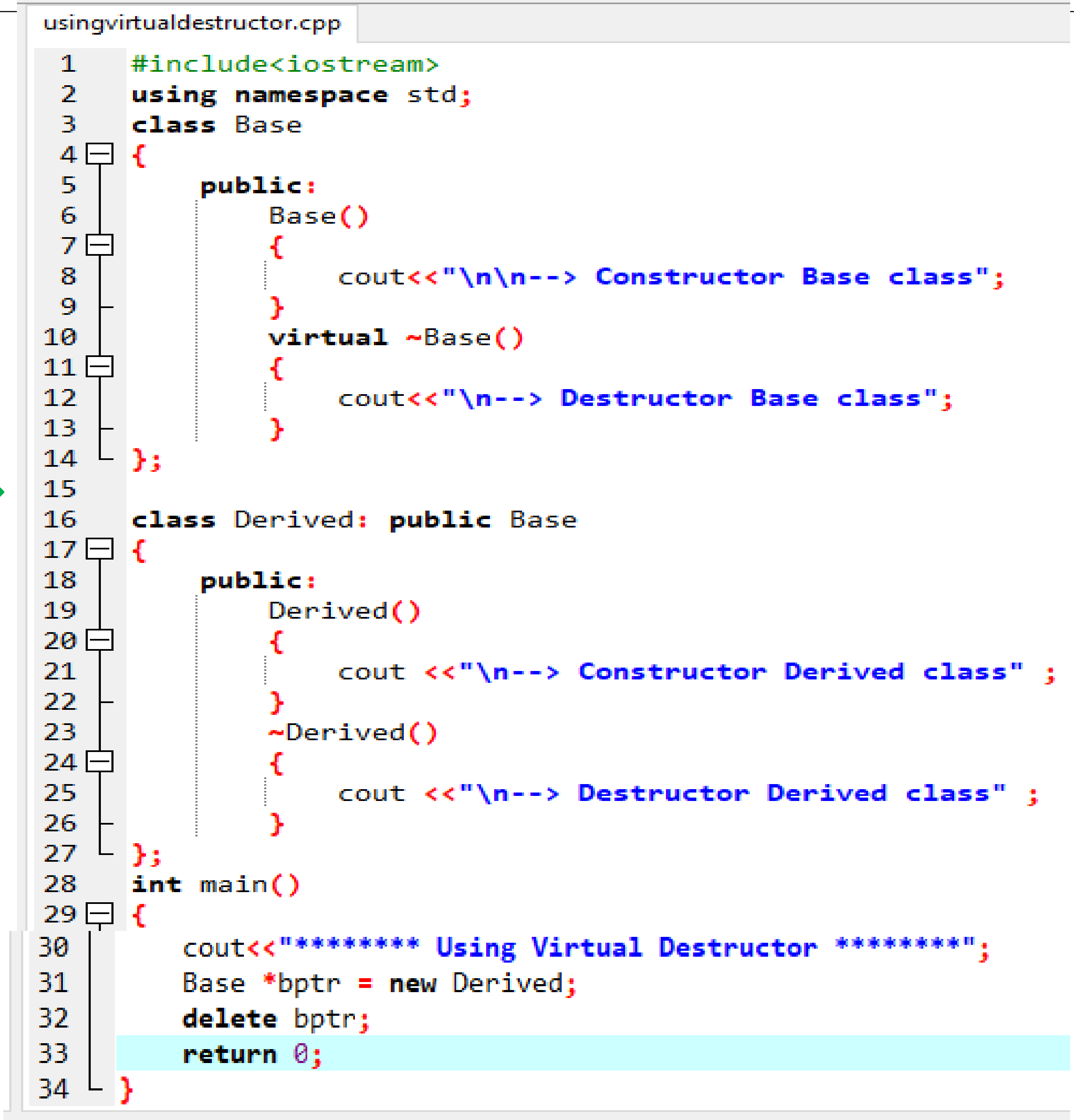
F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\withoutvirtualdestructor....

```
******** Without Using Virtual Destructor ********

--> Constructor Base class
--> Constructor Derived class
--> Destructor Base class
--------------------------------
Process exited after 0.1052 seconds with return value 0
Press any key to continue . . .
```

It is executing the code of Base Class Constructor, Derived Class Constructor and Base Class Destructor

e.g.

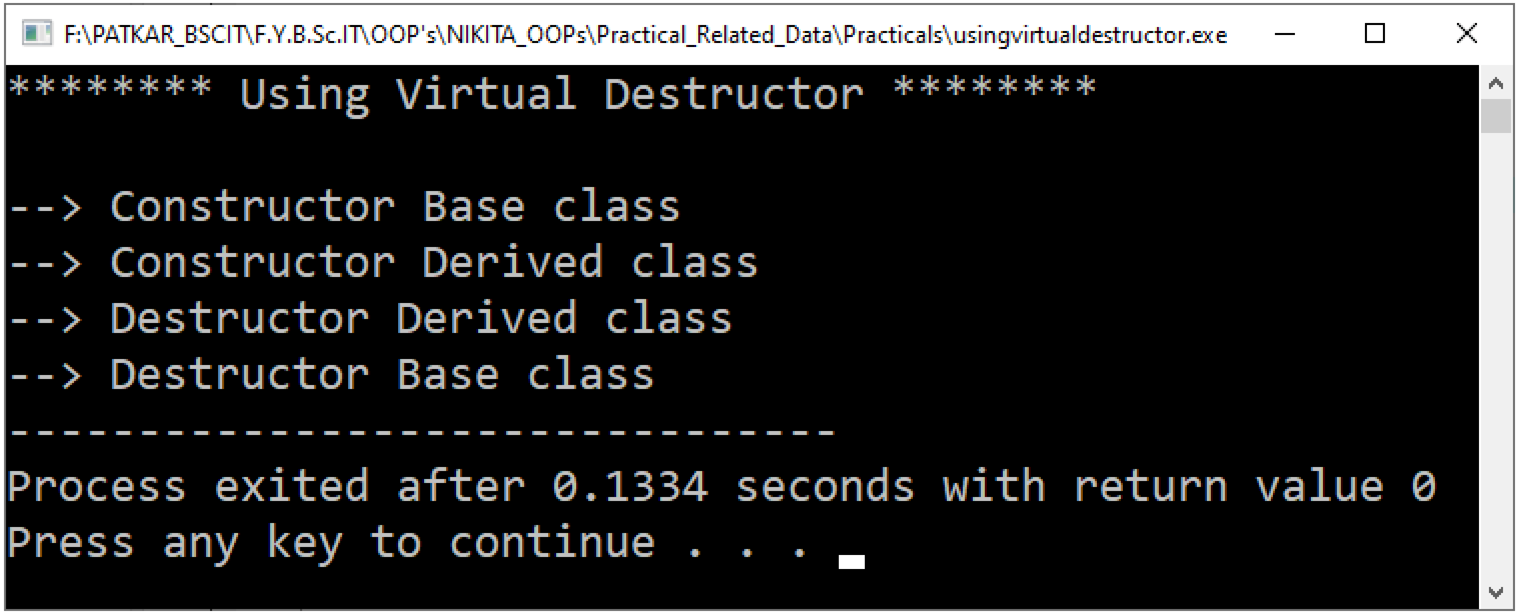
Using Virtual Destructor



usingvirtualdestructor.cpp

```cpp
#include<iostream>
using namespace std;
class Base
{
    public:
        Base()
        {
            cout<<"\n\n--> Constructor Base class";
        }
        virtual ~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()
{
    cout<<"******** Using Virtual Destructor ********";
    Base *bptr = new Derived;
    delete bptr;
    return 0;
}
```

Output :



F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\usingvirtualdestructor.exe

```
********* Using Virtual Destructor *********

--> Constructor Base class
--> Constructor Derived class
--> Destructor Derived class
--> Destructor Base class
--------------------------------

Process exited after 0.1334 seconds with return value 0
Press any key to continue . . .
```

It is executing the code of Base Class Constructor, Derived Class Constructor , Derived Class Destructor and Base Class Destructor

11.
INPUT-OUTPUT AND MANIPULATORS IN C++

Introduction

- C++ uses the concept of stream to make I/O operation fast.
- As we know ,every program go through the process of input-output flow, so it takes some data as input and produce the output based on the output.
- To implement input/output operations, c++ **uses stream and stream classes.**
- C++ I/O system contains a hierarchy of **classes that are used to define various streams to deal with both the console and disk files**. These **classes** are called **stream classes**.

- A stream is a sequence of data, measured in bytes.
- **Streams** are **divided into** ***input stream and output stream.***
- The **source stream** that **provides data to the program** is **called** the **input stream** and the ***destination stream*** that ***receives output from the program*** is ***called*** the ***output stream***.
- The **data in the input stream** can **come from** the **keyboard or any other storage device**.
- The ***data in the output stream*** can ***go to*** the ***screen or any other storage device***

Stream Class

- In C++ there are number of streams classes for defining various streams related with files and for doing input output operations.
- All these classes are defined in the file iostream.h

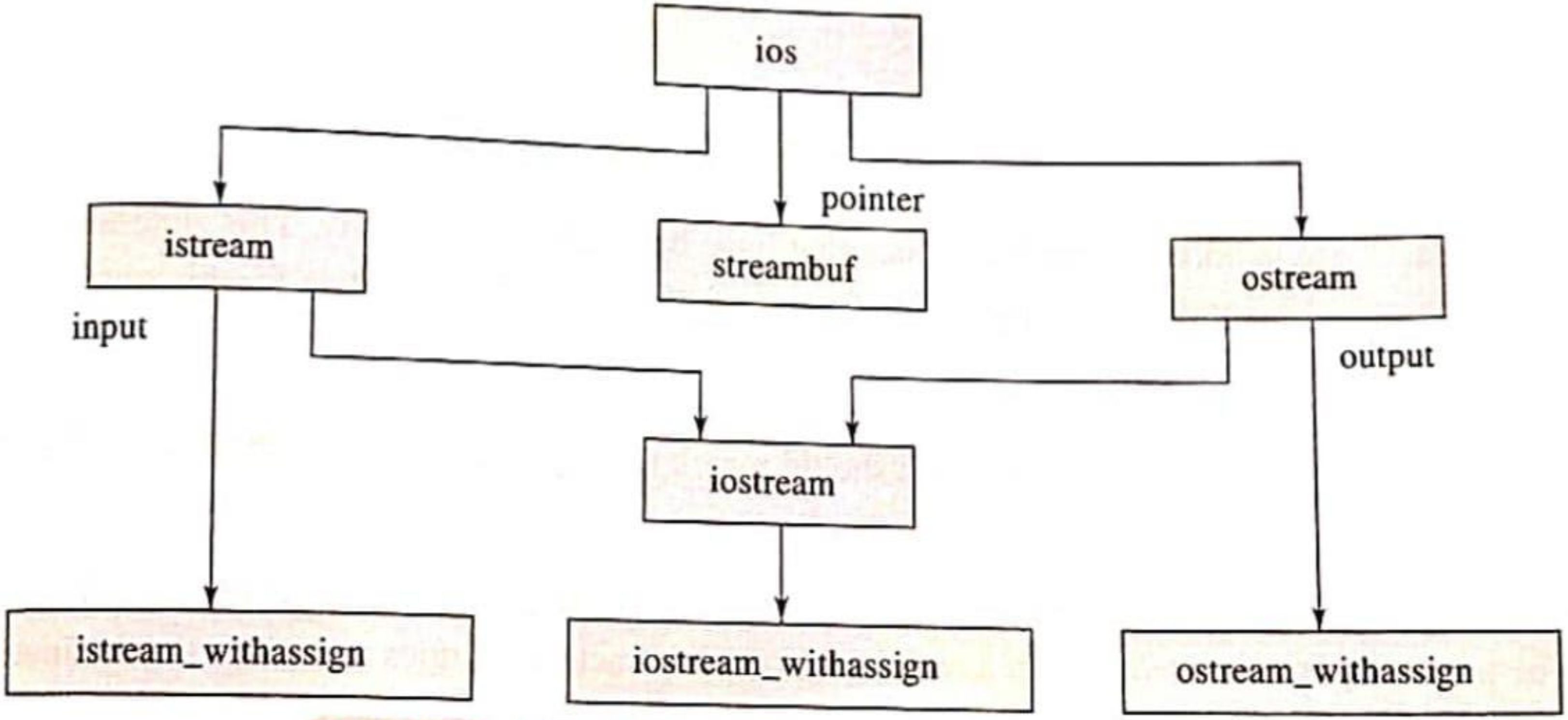


Fig. 10.2 *Stream classes for console I/O operations*

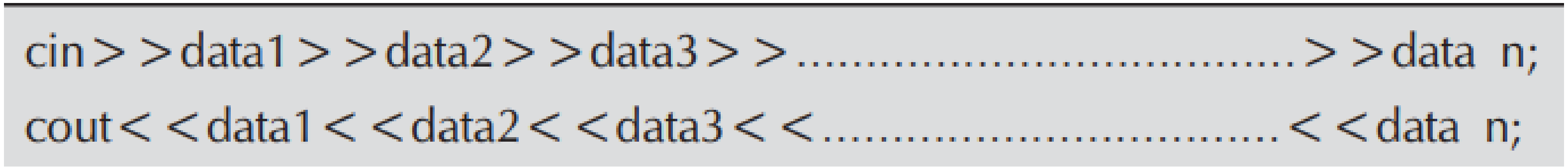
- **ios class** is the **topmost class** in the **stream classes hierarchy**. It is the **base class** for **istream, ostream and streambuf class.**
- **istream, ostream** serves the **base classes** for **iostream class**. The class **istream is used for input** and **ostream for output**.
- Class ios is indirectly to iostream class using istream and ostream. To **avoid the duplicity** of **data and member functions of ios class**, it is **declared as virtual base class** when inheriting in istream and ostream
- The _withassign classes are provided with extra functionality for the assignment operations that's why _withassign classes.

- **ios class**
- It provides **operations common to both input and output**
- It contains a **pointer to a buffer object** (streambuf).
- It has constants and member functions that are necessary for handling formatted input and output operations.
- **istream class**
- It is **derived class** ***of*** **ios** means it inherits all the properties of ios
- It **defines input functions** such as get(),getline()
- **Contains extraction operator >> to read data** from standard input device to memory items.

- **ostream class**
- It is **derived class *of* ios** means it inherits all the properties of ios
- It **defines output functions** such as put() and write()
- **Contains insertion operator << to write data** from memory items to a standard output device.
- **iostream class**
- It **inherits the properties *of* ios, istream and ostream** through multiple inheritance
- It provides the **facility for handling both input and output stream**
- Remaining three classes – istream_withassign, ostream_withassign and iostream_withassign add assignment operator to these classes.

Unformatted Input/Output

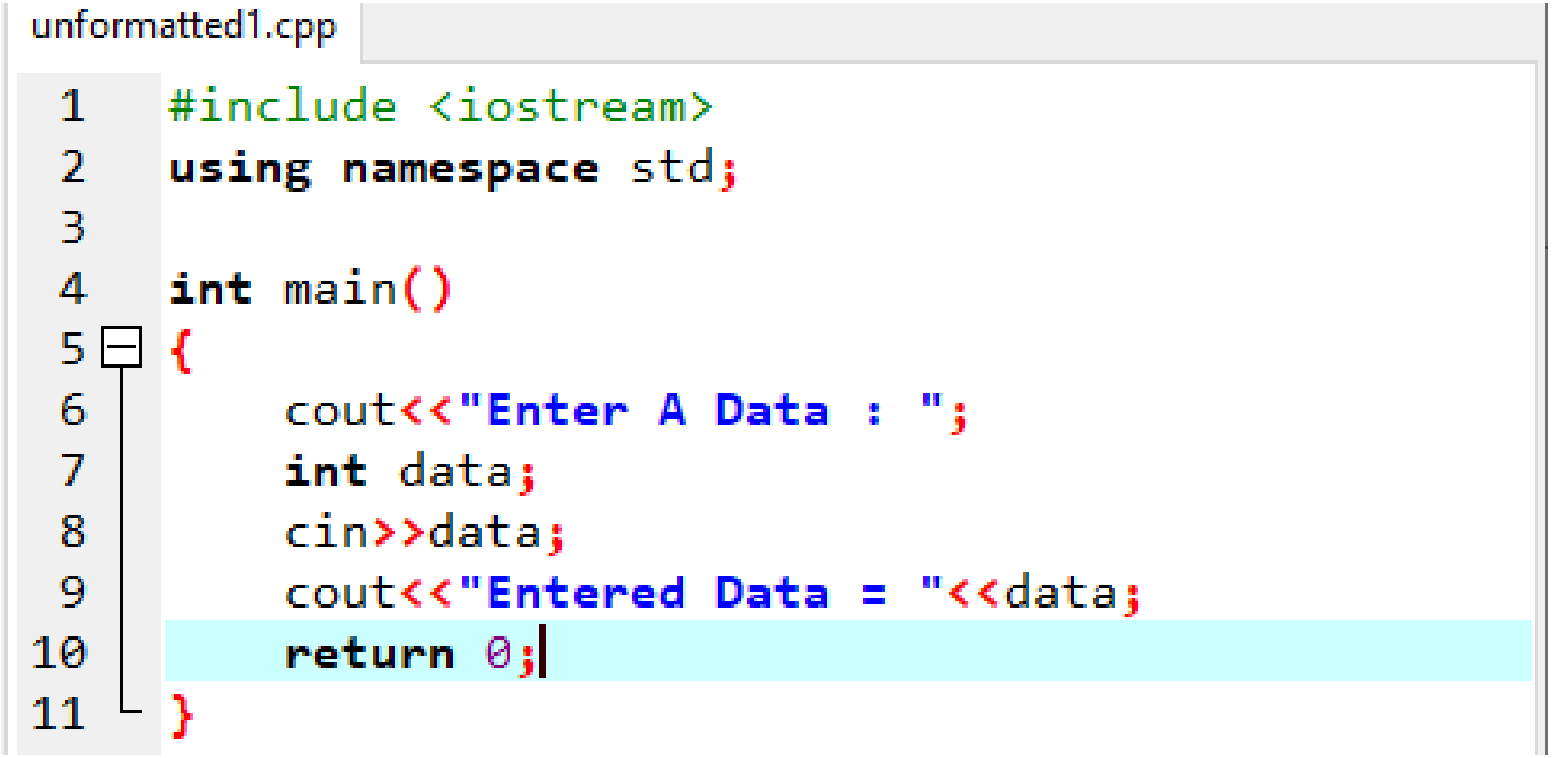
- **Overloaded operators >> and <<**
- We have used the **objects cin and cout** (predefined in the iostream file) **for the input and output of data** of various type. This has been made **possible by overloading the operators >> and <<** to recognize all this basic types.
- The **>> operator is overloaded in the istream** class and **<< is overloaded in the ostream class.**
- General syntax of using data as input and output is as follows :



```
cin>>data1>>data2>>data3>>..................................>>data n;
cout<<data1<<data2<<data3<<................................<<data n;
```

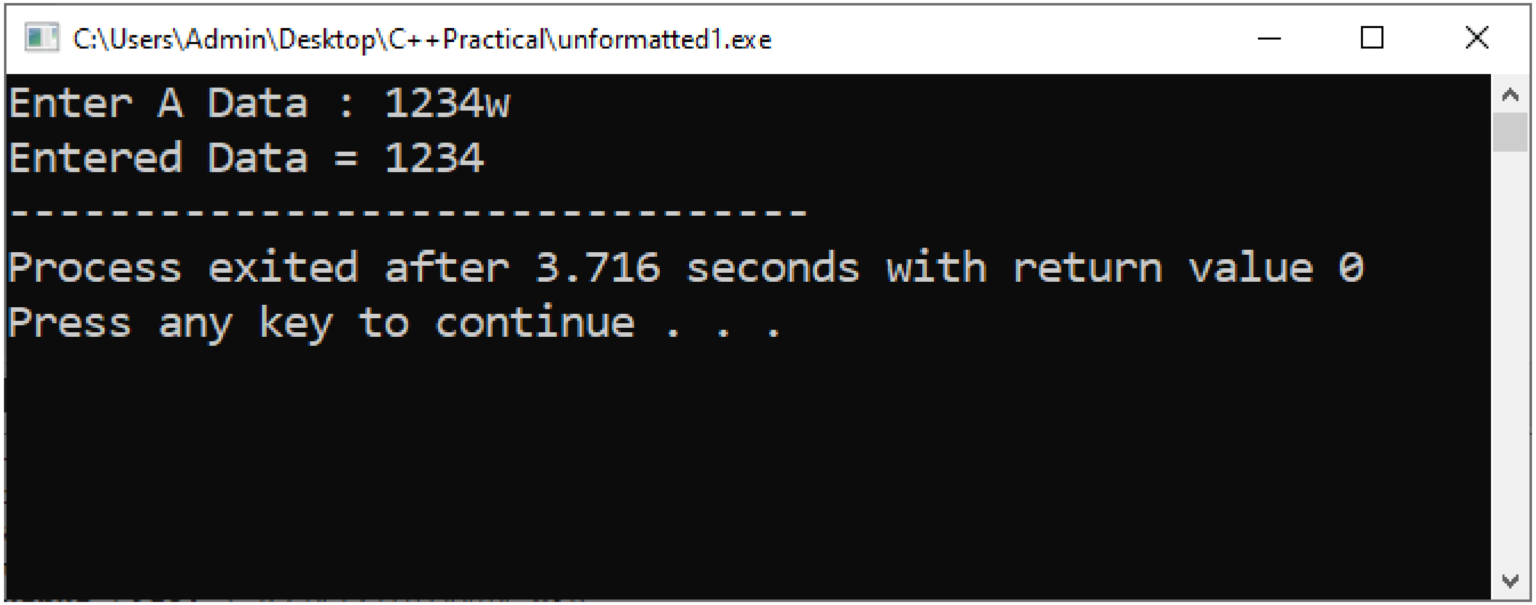
- **NOTE :** The operator reads the data character by character and assign to the indicated location. The **reading for a variable will be terminated** at the e**ncounter of white space or a character that does not match the destination type**.

- E.g.



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

int main()
{
    cout<<"Enter A Data : ";
    int data;
    cin>>data;
    cout<<"Entered Data = "<<data;
    return 0;
}
```

- Output:



```
Enter A Data : 1234w
Entered Data = 1234
--------------------------------
Process exited after 3.716 seconds with return value 0
Press any key to continue . . .
```

get() and put() function

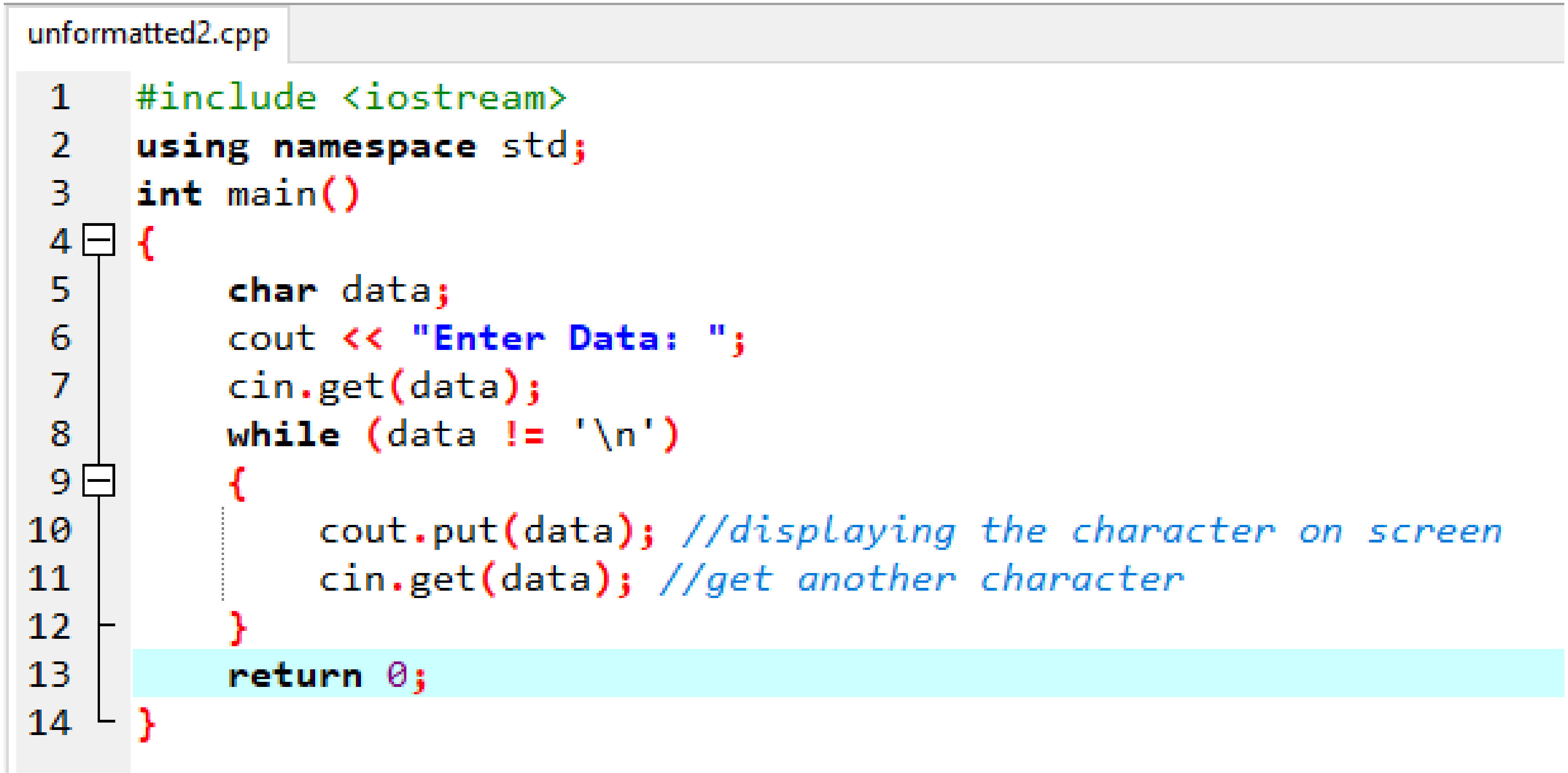
get()

- **get() is a member function of the input stream class istream** and it is used to **read a single character** from the input device.
- There are two types of get() functions.
- Both get(char*) and get() prototype can be used to fetch a character **including blank space, tab and newline character**.

put()

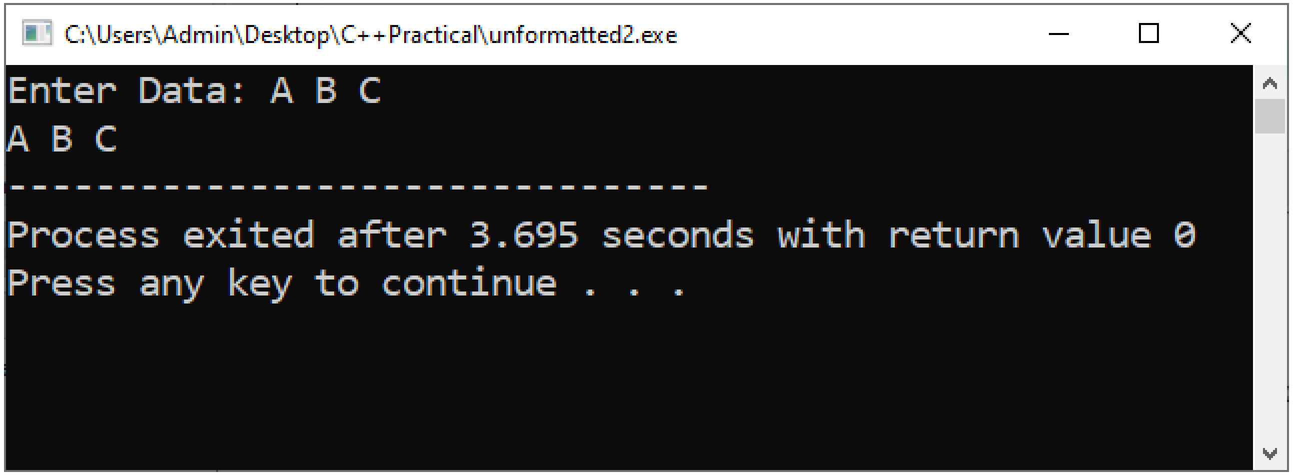
- The **put() is a member function of the output stream class ostream** and it is used to **write a single character** to the output device.

- E.g.



```cpp
#include <iostream>
using namespace std;
int main()
{
    char data;
    cout << "Enter Data: ";
    cin.get(data);
    while (data != '\n')
    {
        cout.put(data); //displaying the character on screen
        cin.get(data); //get another character
    }
    return 0;
}
```

- Output:



```
Enter Data: A B C
A B C
--------------------------------
Process exited after 3.695 seconds with return value 0
Press any key to continue . . .
```

getline() and write() function

- A line of text can be **read or display effectively** using the line oriented **input/output functions getline() and write()**

getline()

- The **getline()** function **reads a whole line of text ends with a newline character**. The function can be invoked by using the object cin as follows:

cin.getline(line, size);

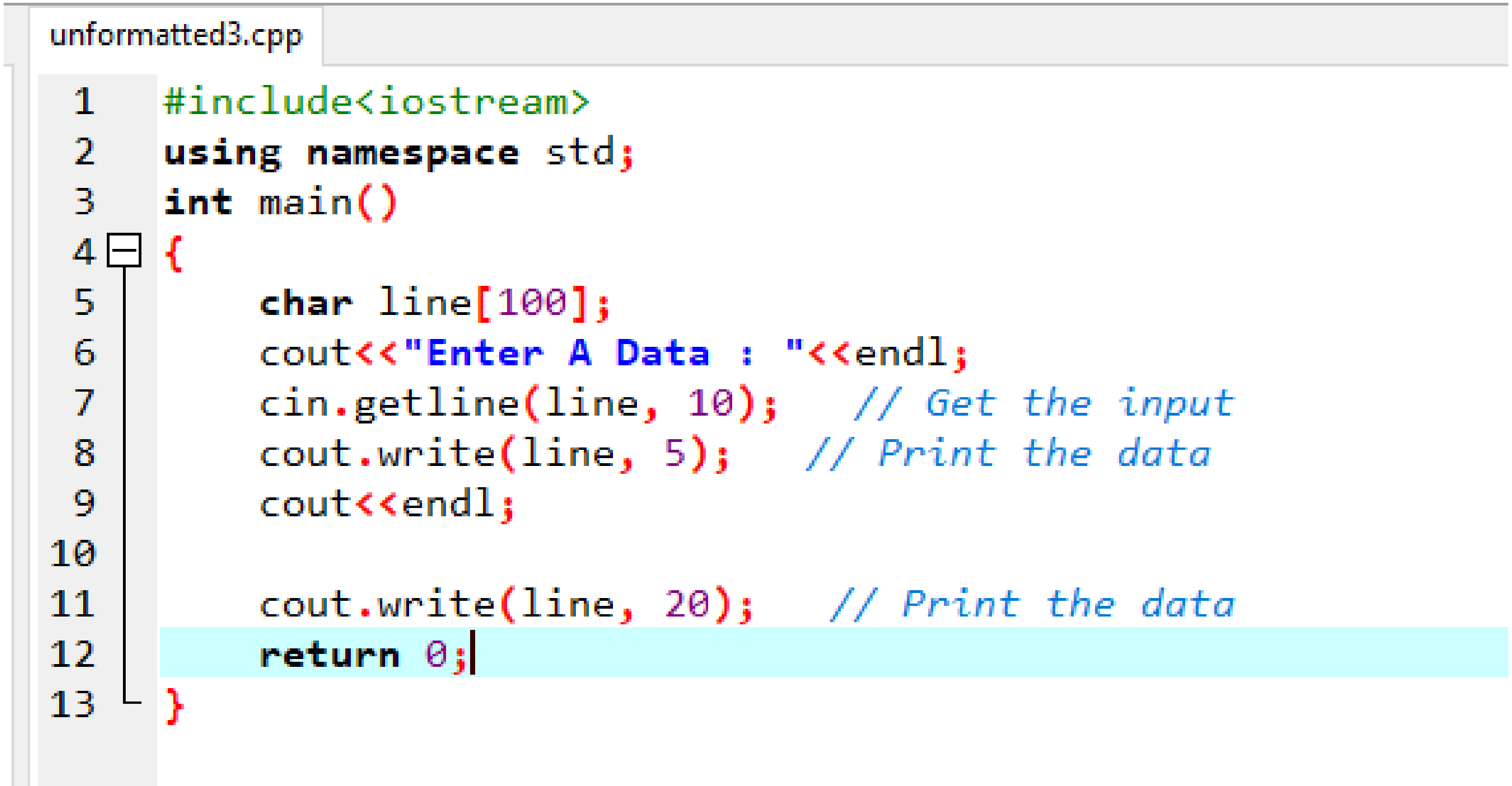
- The **reading is terminated** as soon as **either the new line character '\n' is encountered or size-1 character are read.**
- In this function, the *blank spaces contained in the string are also taken into a count*.

write()

- The **write() function displays the entire line in one go** and its syntax is similar to the getline() function only that here **cout object** is used to invoke it as follows:

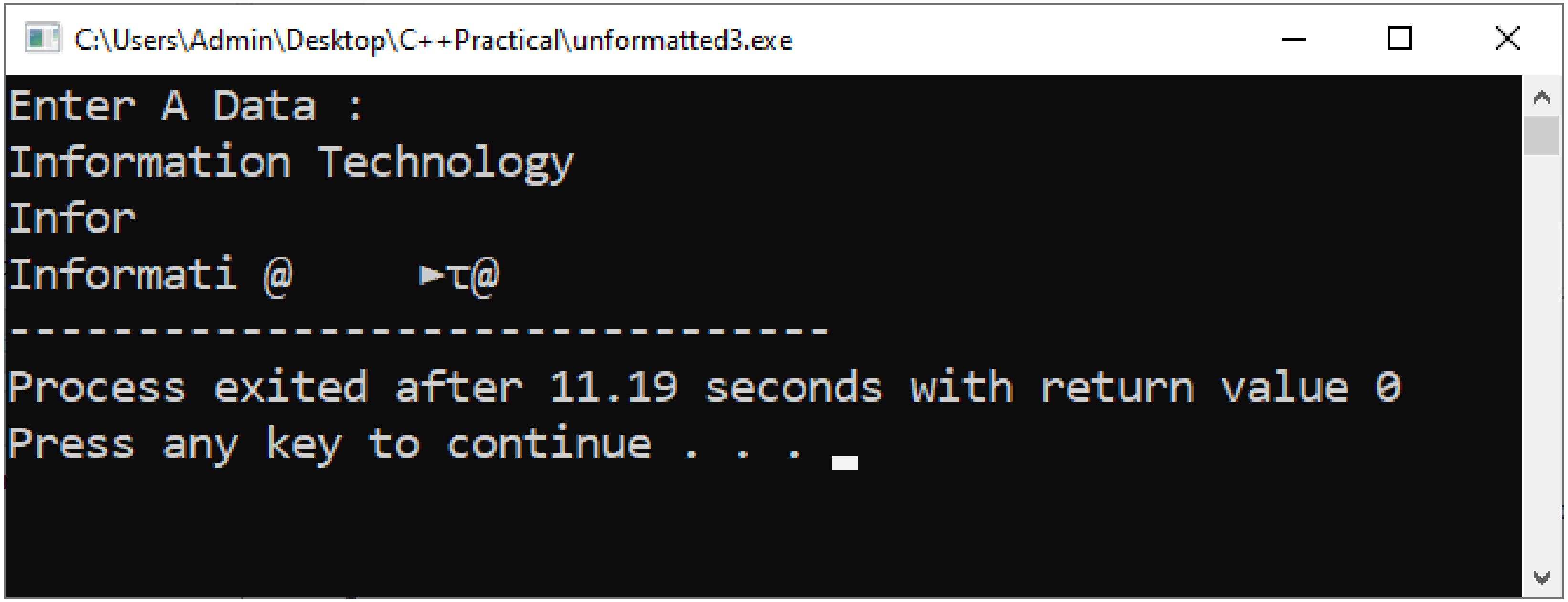
cout. write(line, size);

- E.g.



```cpp
#include<iostream>
using namespace std;
int main()
{
	char line[100];
	cout<<"Enter A Data : "<<endl;
	cin.getline(line, 10);   // Get the input
	cout.write(line, 5);   // Print the data
	cout<<endl;

	cout.write(line, 20);   // Print the data
	return 0;
}
```

- Output:



```
Enter A Data :
Information Technology
Infor
Informati @     ►τ@
--------------------------------

Process exited after 11.19 seconds with return value 0
Press any key to continue . . .
```

Formatted Input /Output Operations

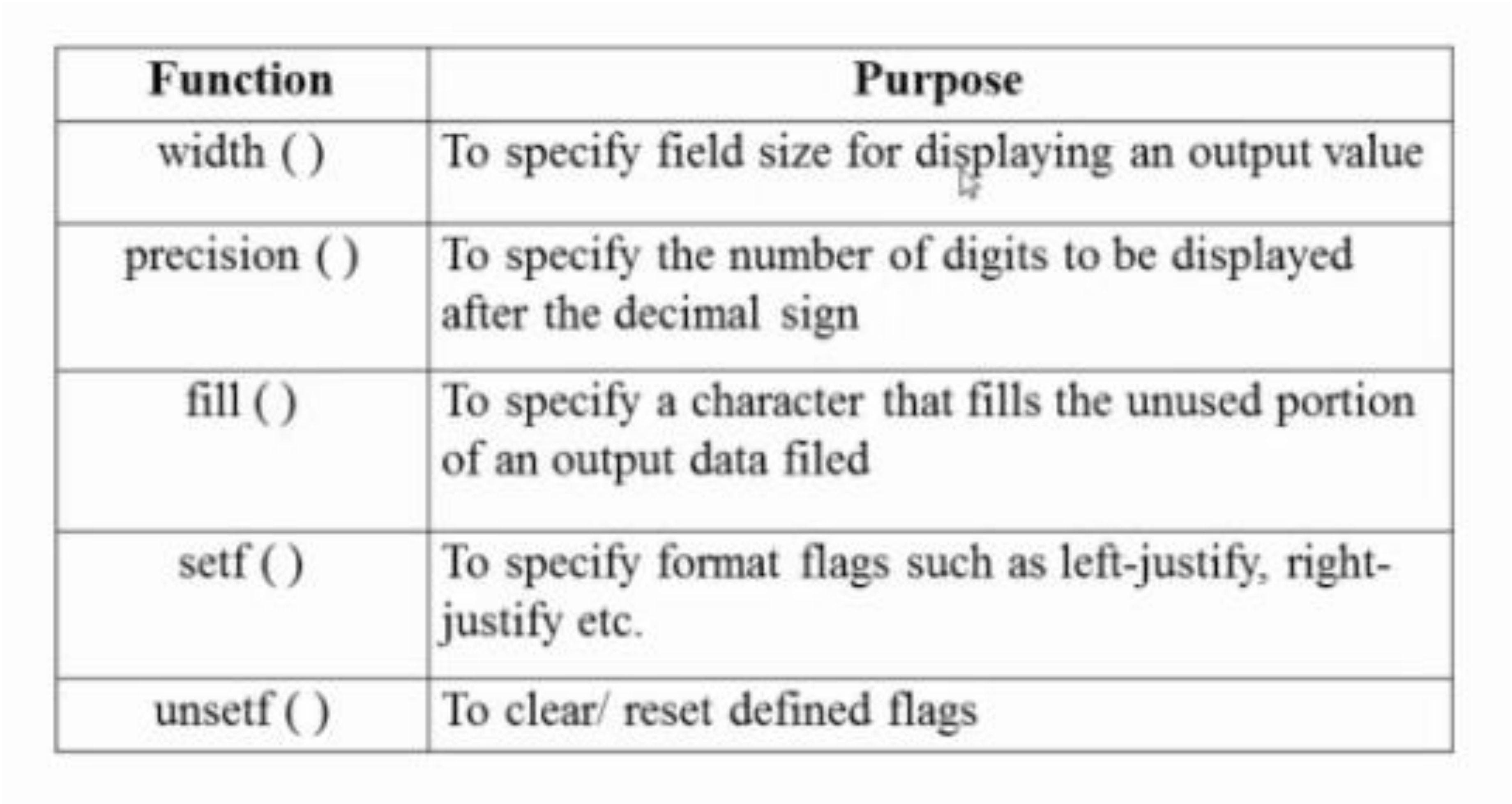
- C++ supports a variety of features **to perform input or output in different formats**. They include following features:

1. Functions and flags defined by ios class.
2. Use of manipulators (built-in)
3. User defined manipulators

➢ **ios class functions and flags**

- The **ios stream class contains** a **large number of member functions** that **help** us **in formatting the output in a number of ways**.
- All the functions are called using the built-in object **cout**

- The functions are as follows:



Function	Purpose
width ( )	To specify field size for displaying an output value
precision ( )	To specify the number of digits to be displayed after the decimal sign
fill ( )	To specify a character that fills the unused portion of an output data filed
setf ( )	To specify format flags such as left-justify, right-justify etc.
unsetf ( )	To clear/ reset defined flags

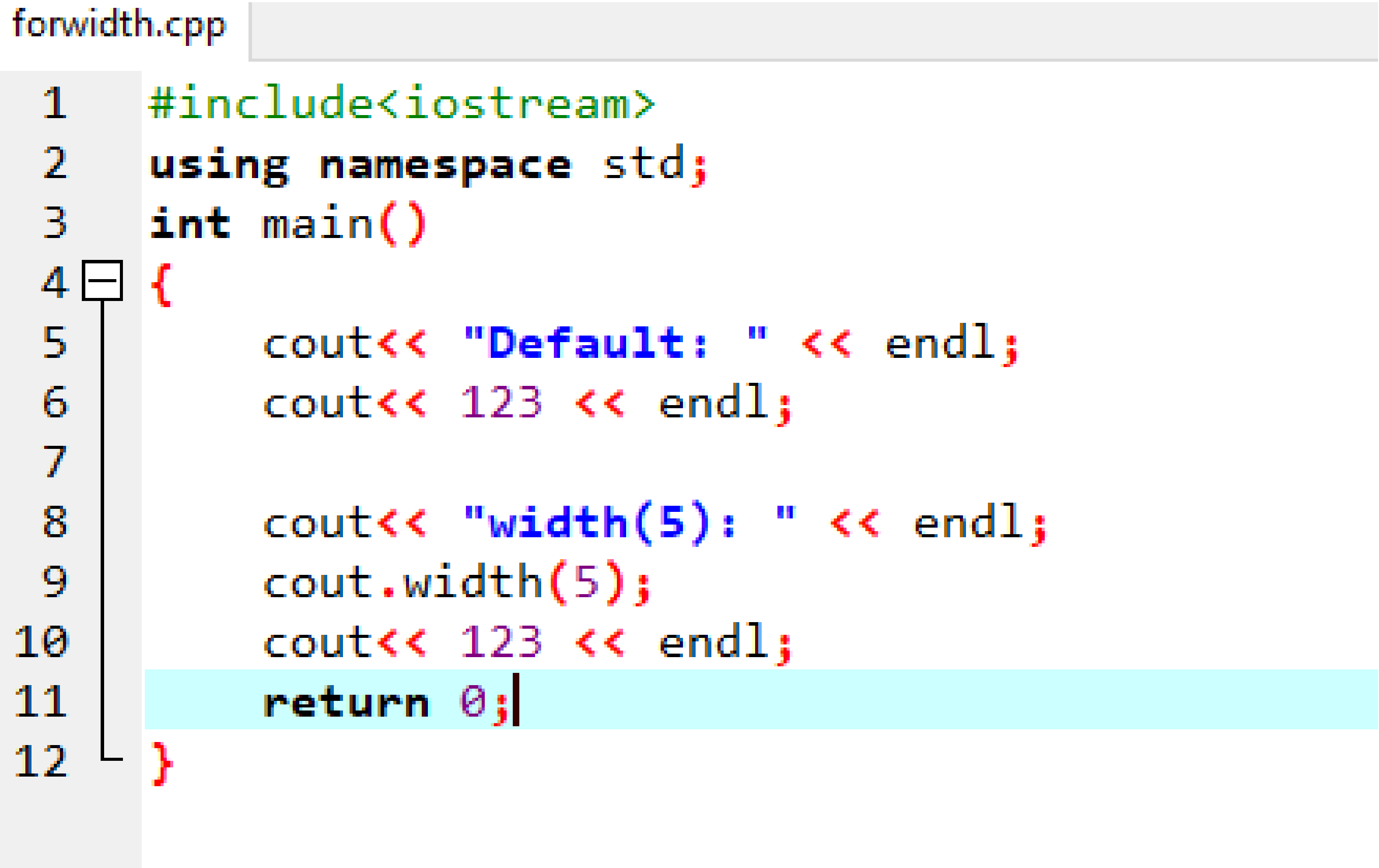
❖width()

- It is a member function of the ios class.
- It **specifies the required number of field size** to be **used while displaying the output values**
- It commonly accessed **using the cout object**
- Syntax:

cout.width(w);

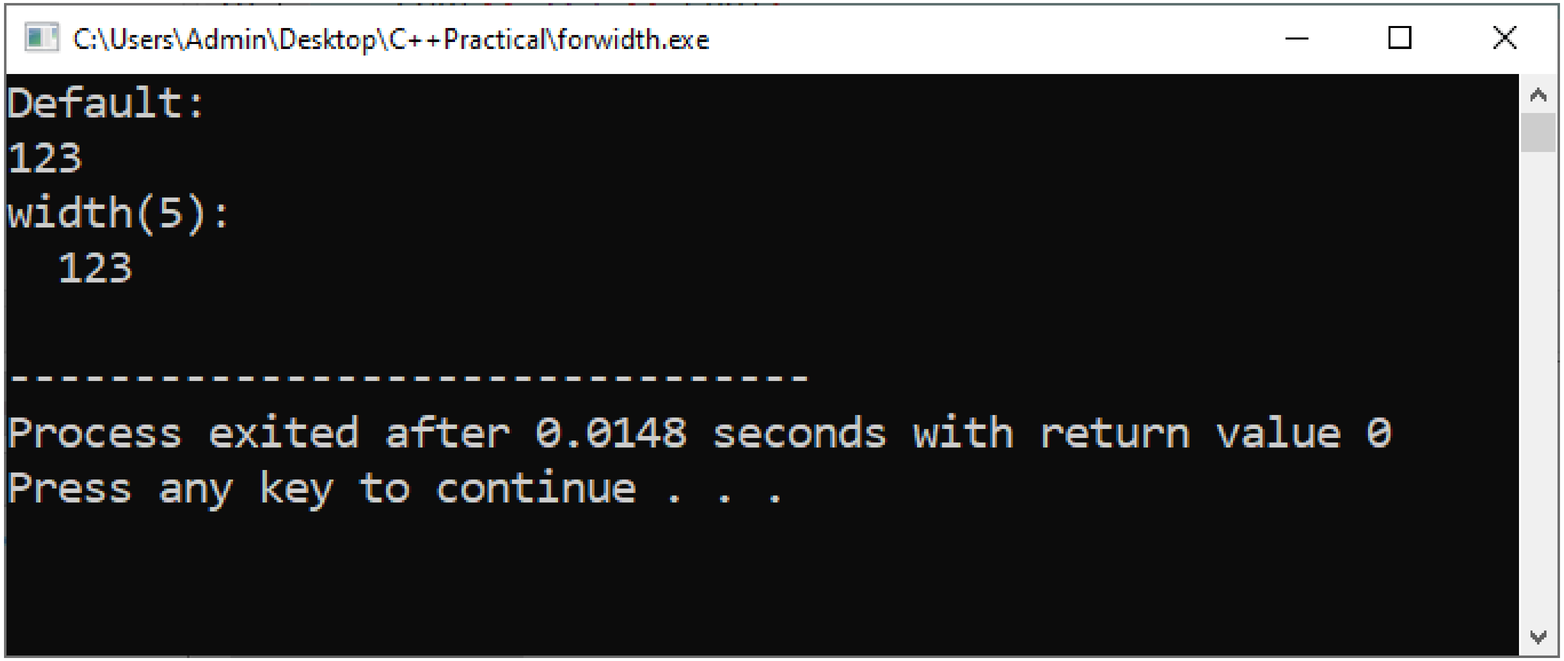
- Where w is the field width, i.e. number of columns required for displaying output.
- The **output will** be printed at the **right end of the field.**

- E.g.



```
#include<iostream>
using namespace std;
int main()
{
    cout<< "Default: " << endl;
    cout<< 123 << endl;

    cout<< "width(5): " << endl;
    cout.width(5);
    cout<< 123 << endl;
    return 0;
}
```

- Output:



```
Default:
123
width(5):
  123

--------------------------------
Process exited after 0.0148 seconds with return value 0
Press any key to continue . . .
```

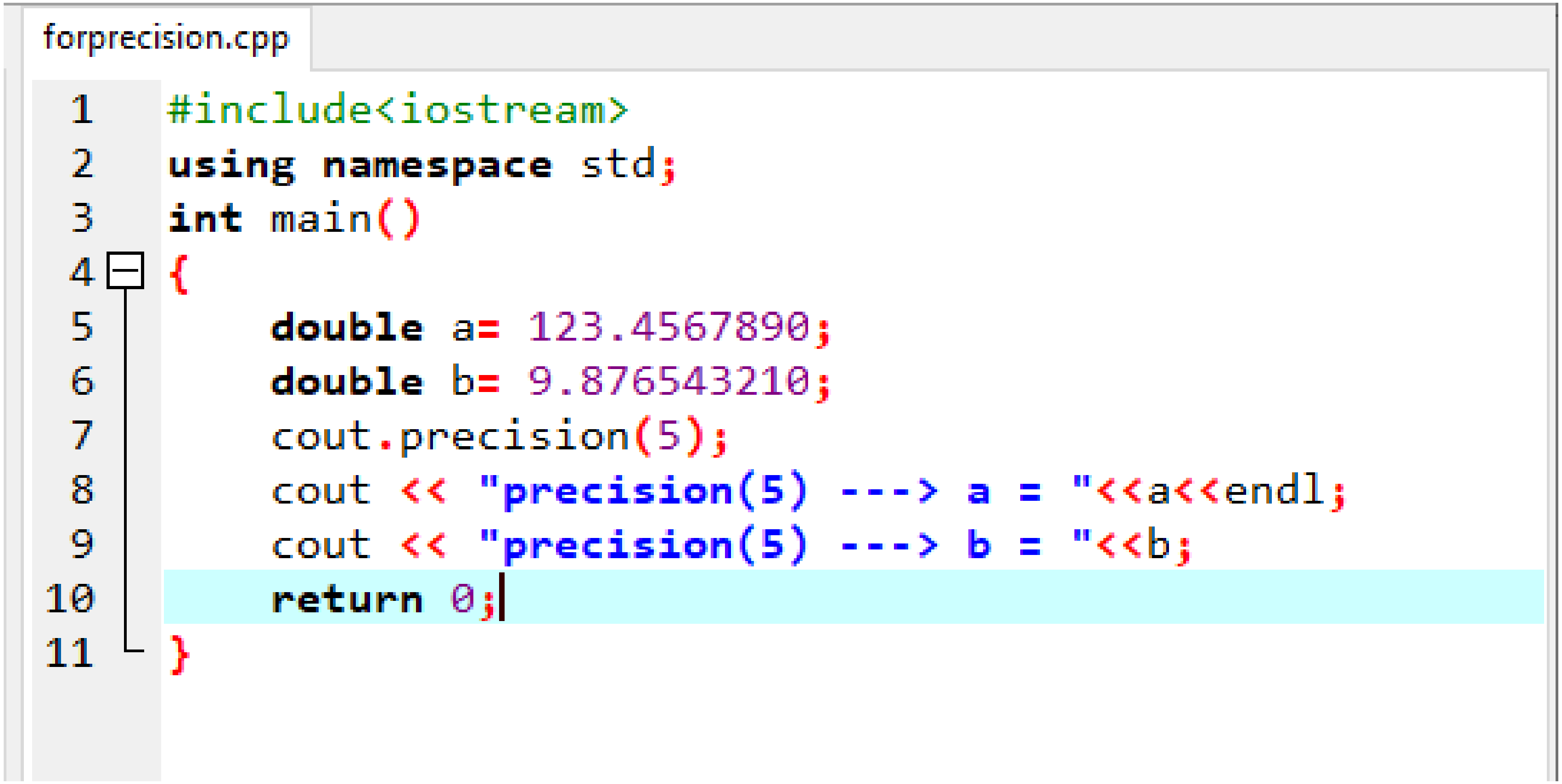
precision()

- It is a member function of the ios class.
- It **specifies the number digits to be displayed after the decimal point**
- It commonly accessed **using the cout object**
- Syntax:

cout.precision(p);

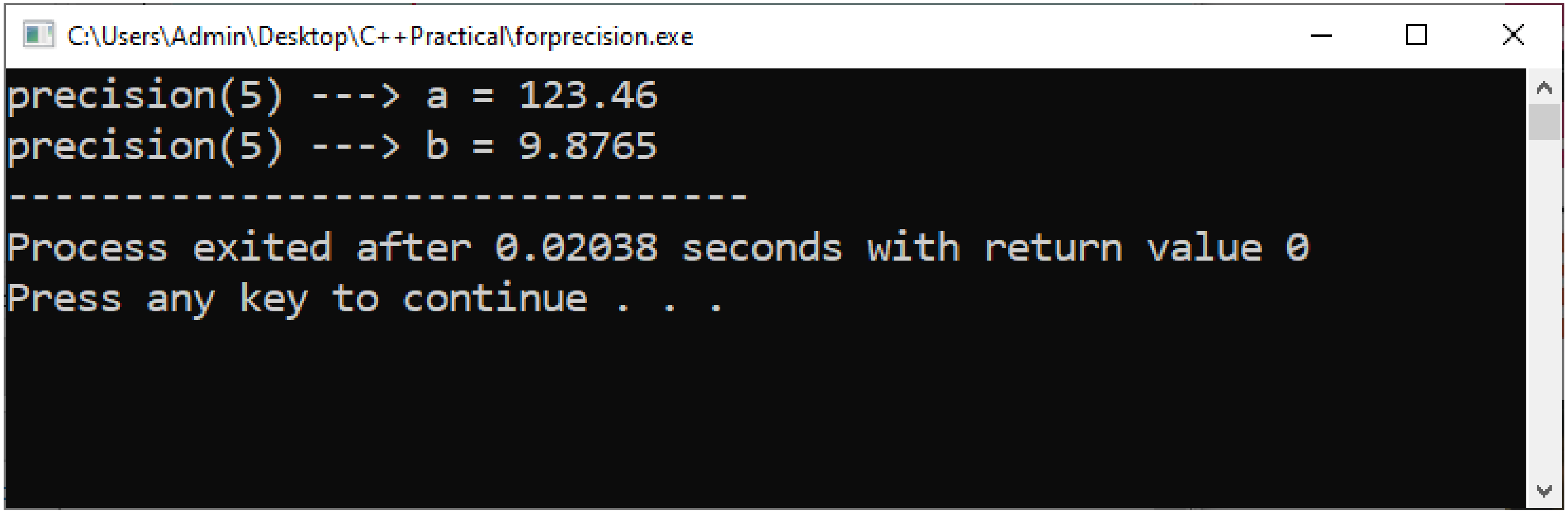
- Where p is the number of digits to the right of the decimal point.

- E.g.



```cpp
#include<iostream>
using namespace std;
int main()
{
    double a= 123.4567890;
    double b= 9.876543210;
    cout.precision(5);
    cout << "precision(5) ---> a = "<<a<<endl;
    cout << "precision(5) ---> b = "<<b;
    return 0;
}
```

- Output:



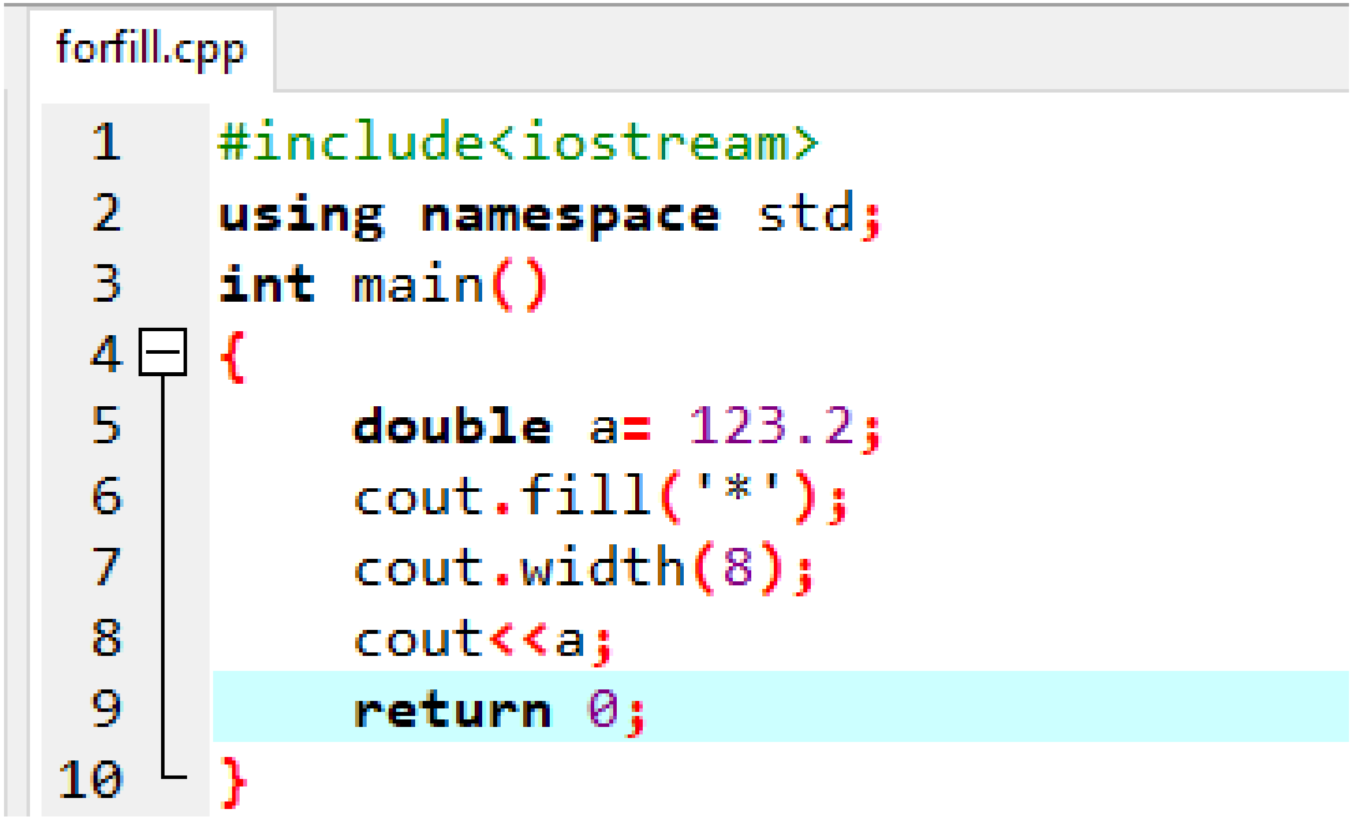
```
precision(5) ---> a = 123.46
precision(5) ---> b = 9.8765
--------------------------------
Process exited after 0.02038 seconds with return value 0
Press any key to continue . . .
```

❖fill()

- It is a member function of the ios class.
- It **specifies a character that is used to fill the unused area of field**
- The **unused area of field *width*** are filled with ***white spaces, by default***.
- It commonly accessed **using the cout object**
- Syntax:

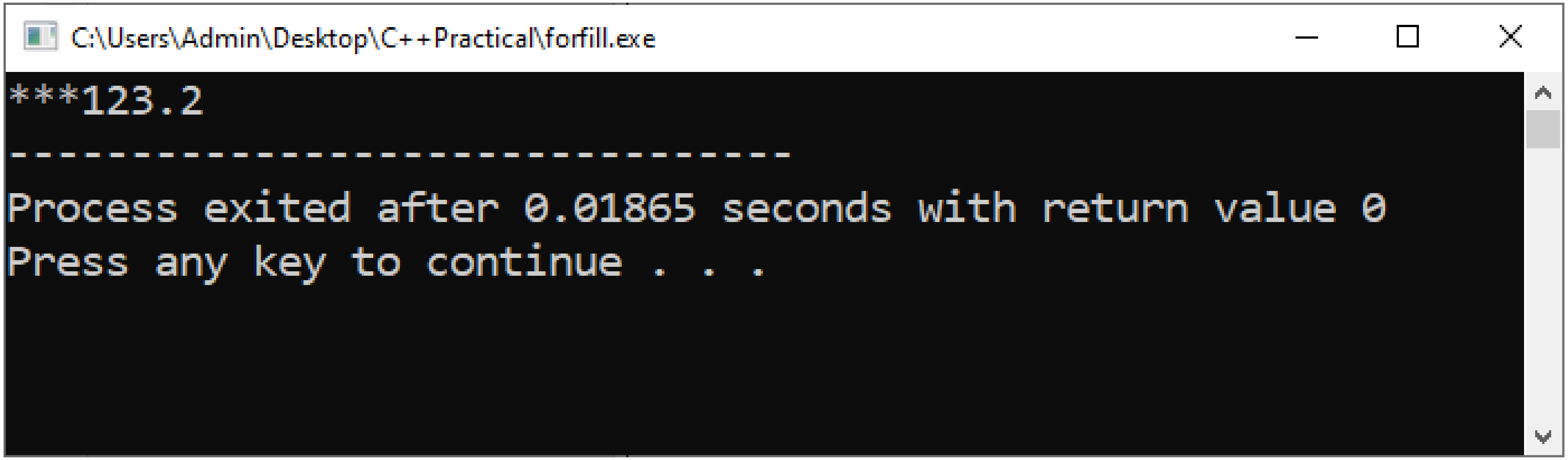
cout.fill(ch);

- E.g.



```cpp
#include<iostream>
using namespace std;
int main()
{
	double a= 123.2;
	cout.fill('*');
	cout.width(8);
	cout<<a;
	return 0;
}
```

- Output:



```
***123.2
--------------------------------
Process exited after 0.01865 seconds with return value 0
Press any key to continue . . .
```

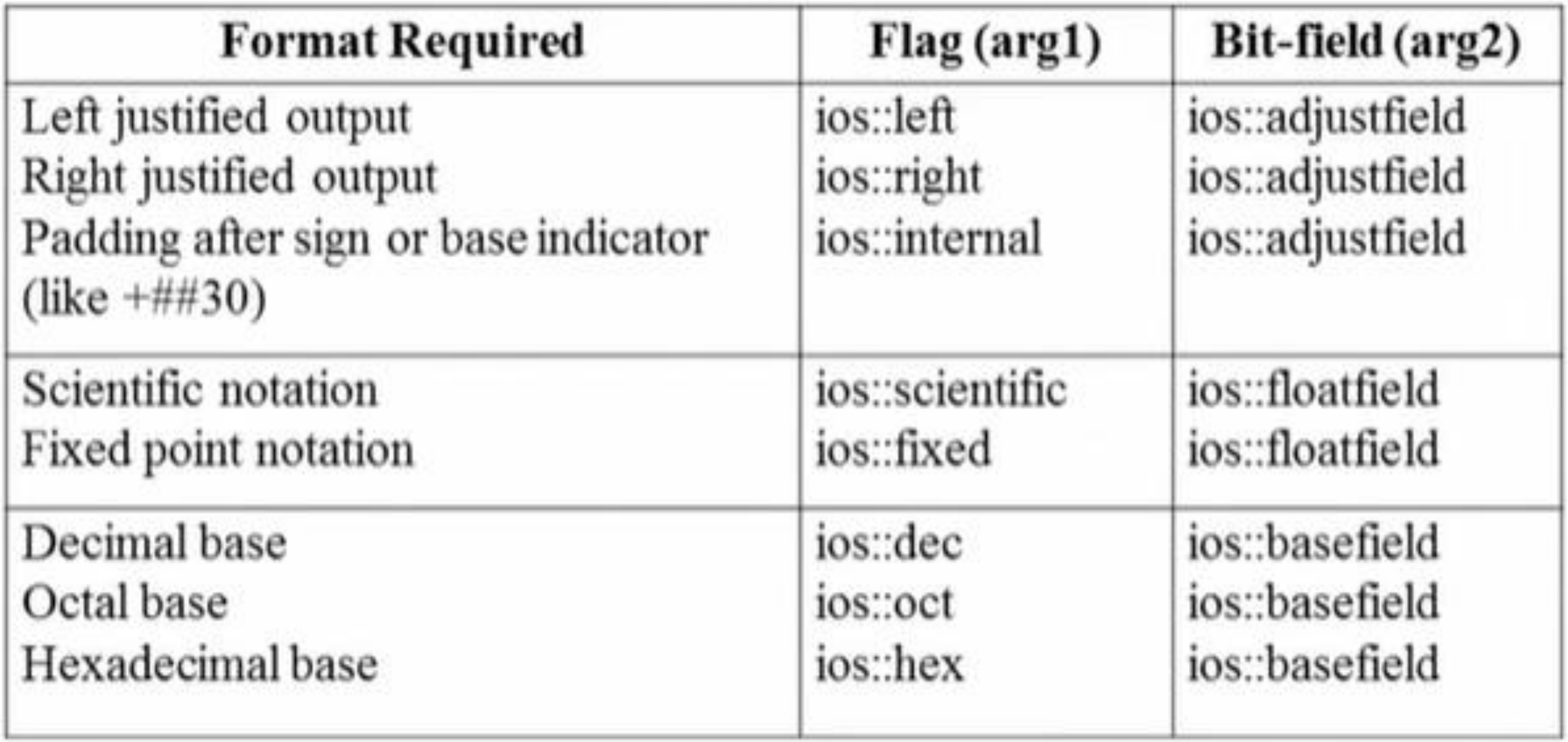
❖Formatting Flags, Bit-Fields setf()

- The setf() is a member function of ios class
- It is used to set flags and bit fields that control the output
- The setf() stands for set flags
- It commonly accessed **using the cout object**
- Syntax:

cout.setf(arg1,arg2);

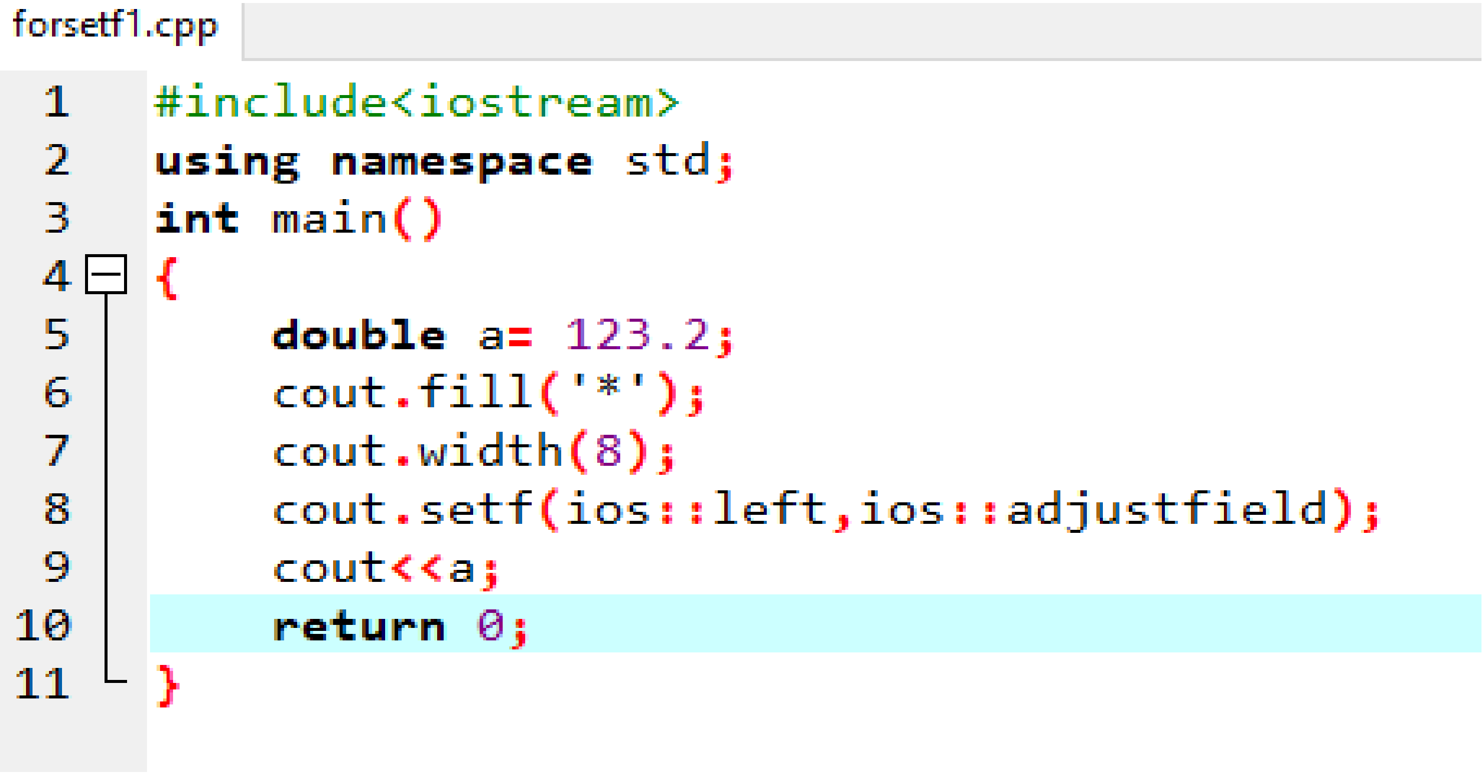
- The arg1 is one of the formatting flags defined in the class ios. It specifies the **format action required for the output.**
- The arg2 known as the field **specifies the group to which the formatting flag belongs**.

- The following table displays the flags and bit fields for set() function



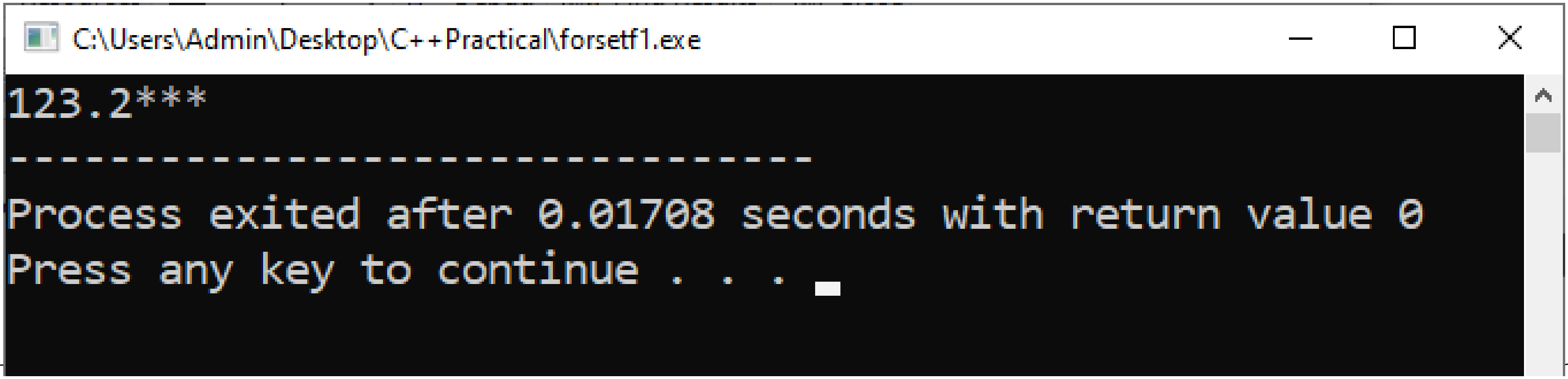
Format Required	Flag (arg1)	Bit-field (arg2)
Left justified output	ios::left	ios::adjustfield
Right justified output	ios::right	ios::adjustfield
Padding after sign or base indicator (like +##30)	ios::internal	ios::adjustfield
Scientific notation	ios::scientific	ios::floatfield
Fixed point notation	ios::fixed	ios::floatfield
Decimal base	ios::dec	ios::basefield
Octal base	ios::oct	ios::basefield
Hexadecimal base	ios::hex	ios::basefield

- E.g.



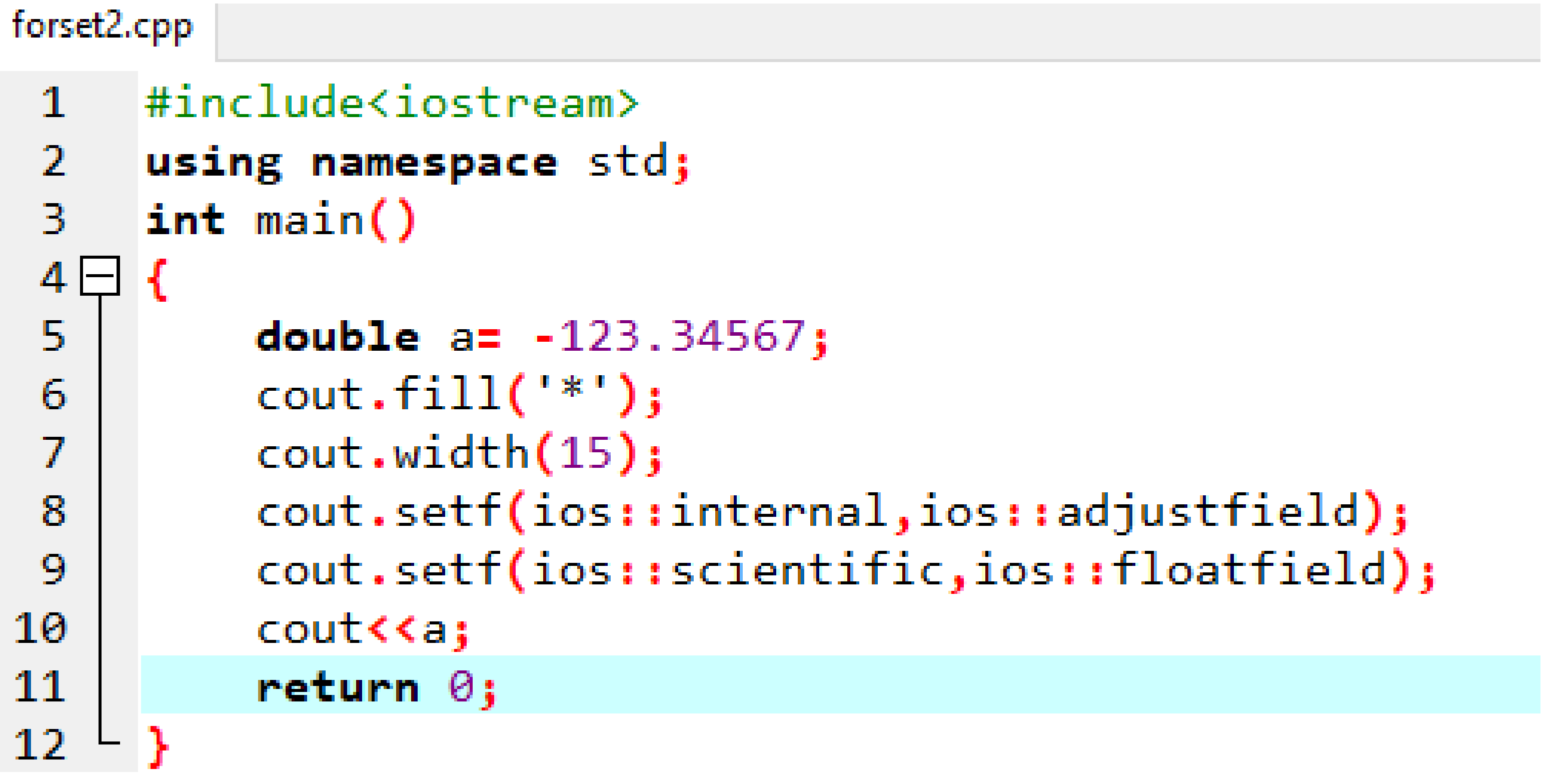
```cpp
#include<iostream>
using namespace std;
int main()
{
	double a= 123.2;
	cout.fill('*');
	cout.width(8);
	cout.setf(ios::left,ios::adjustfield);
	cout<<a;
	return 0;
}
```

- Output:



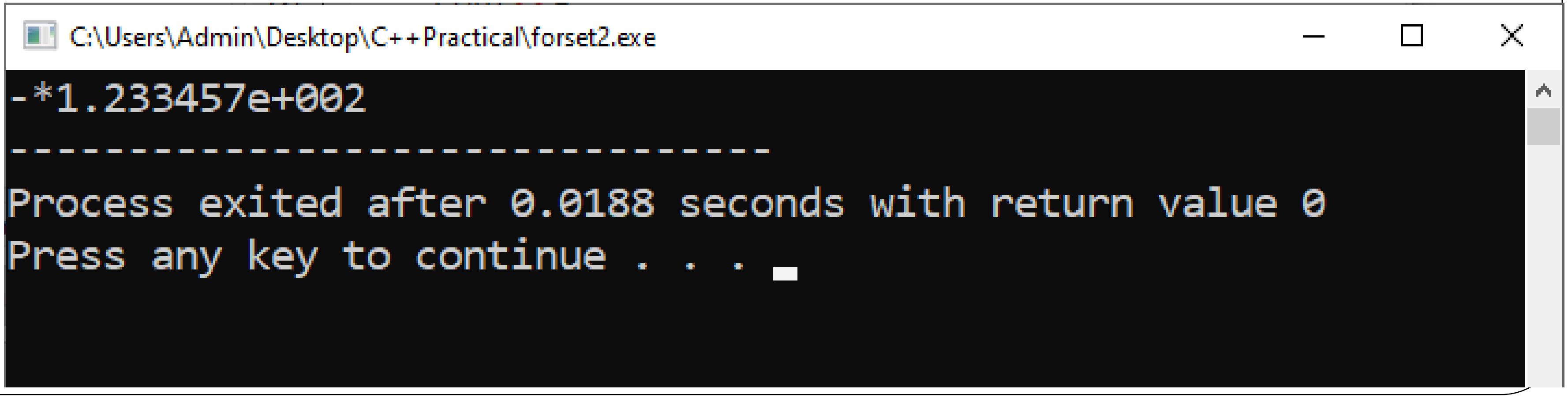
```
123.2***
--------------------------------
Process exited after 0.01708 seconds with return value 0
Press any key to continue . . .
```

- E.g.



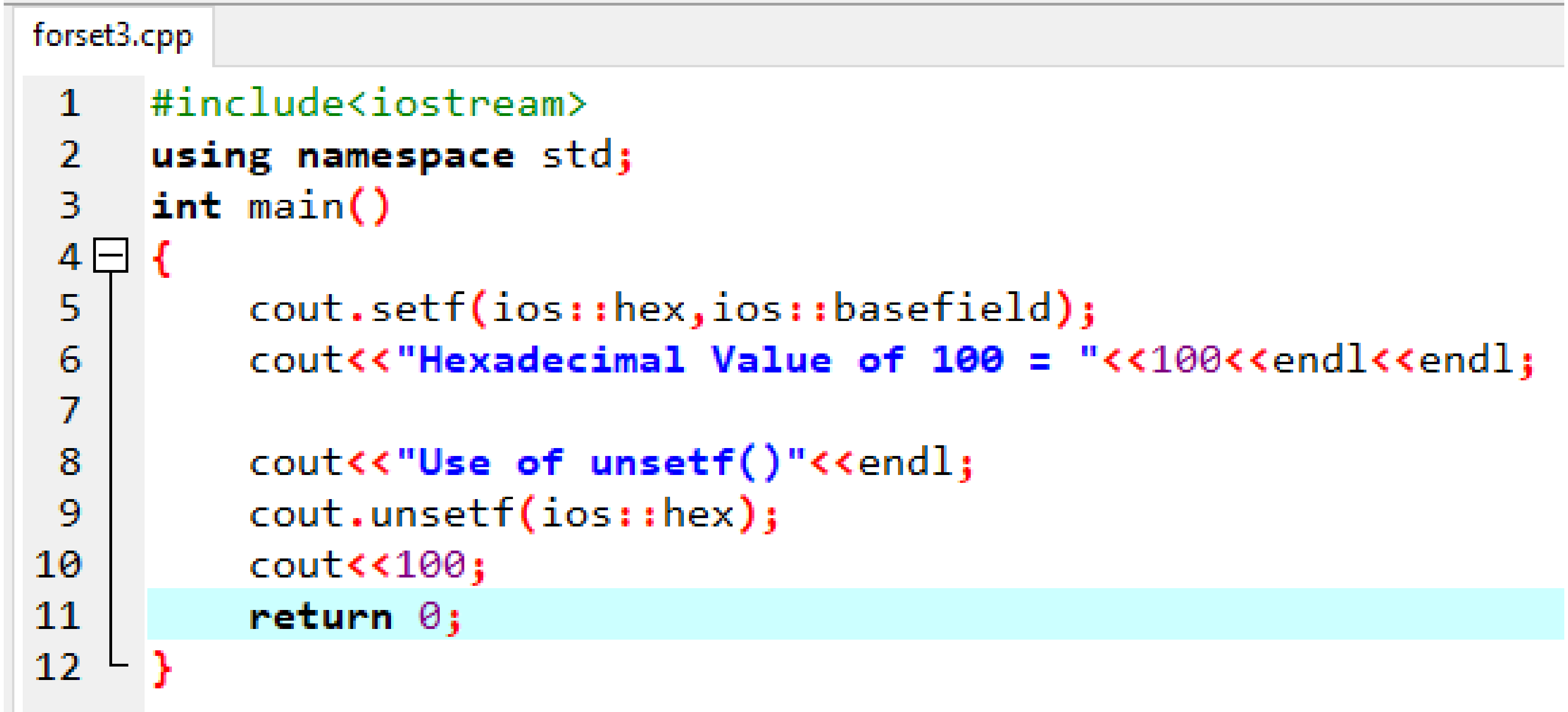
```
#include<iostream>
using namespace std;
int main()
{
	double a= -123.34567;
	cout.fill('*');
	cout.width(15);
	cout.setf(ios::internal,ios::adjustfield);
	cout.setf(ios::scientific,ios::floatfield);
	cout<<a;
	return 0;
}
```

- Output:



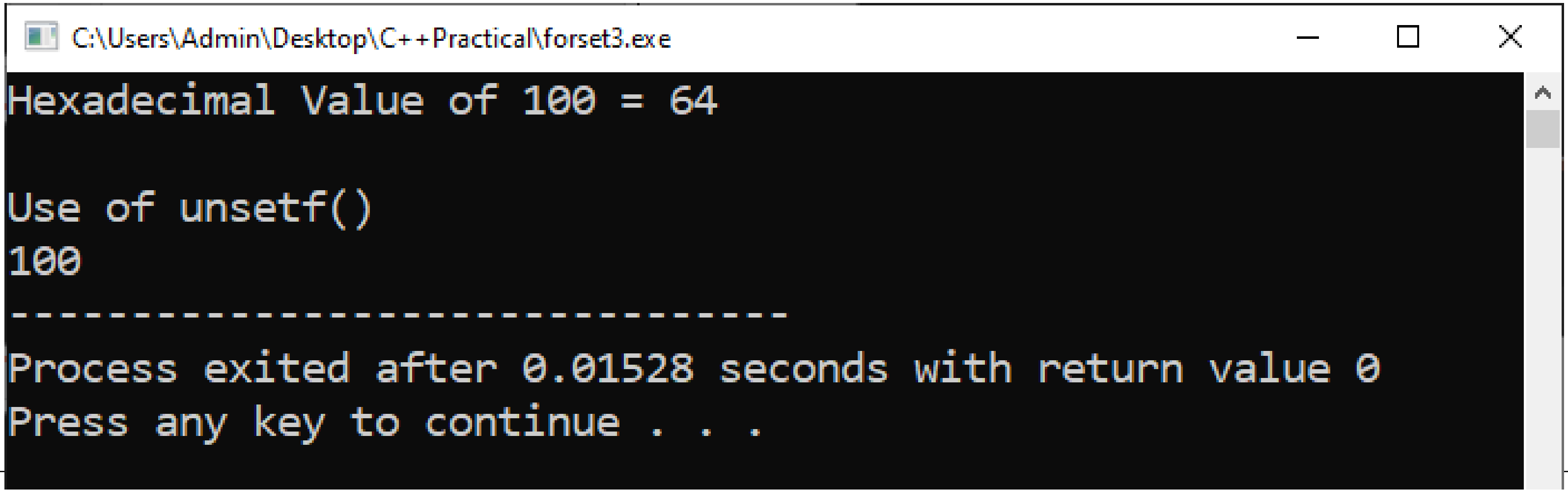
```
-*1.233457e+002
--------------------------------
Process exited after 0.0188 seconds with return value 0
Press any key to continue . . .
```

- E.g.



```cpp
#include<iostream>
using namespace std;
int main()
{
	cout.setf(ios::hex,ios::basefield);
	cout<<"Hexadecimal Value of 100 = "<<100<<endl<<endl;

	cout<<"Use of unsetf()"<<endl;
	cout.unsetf(ios::hex);
	cout<<100;
	return 0;
}
```

- Output:



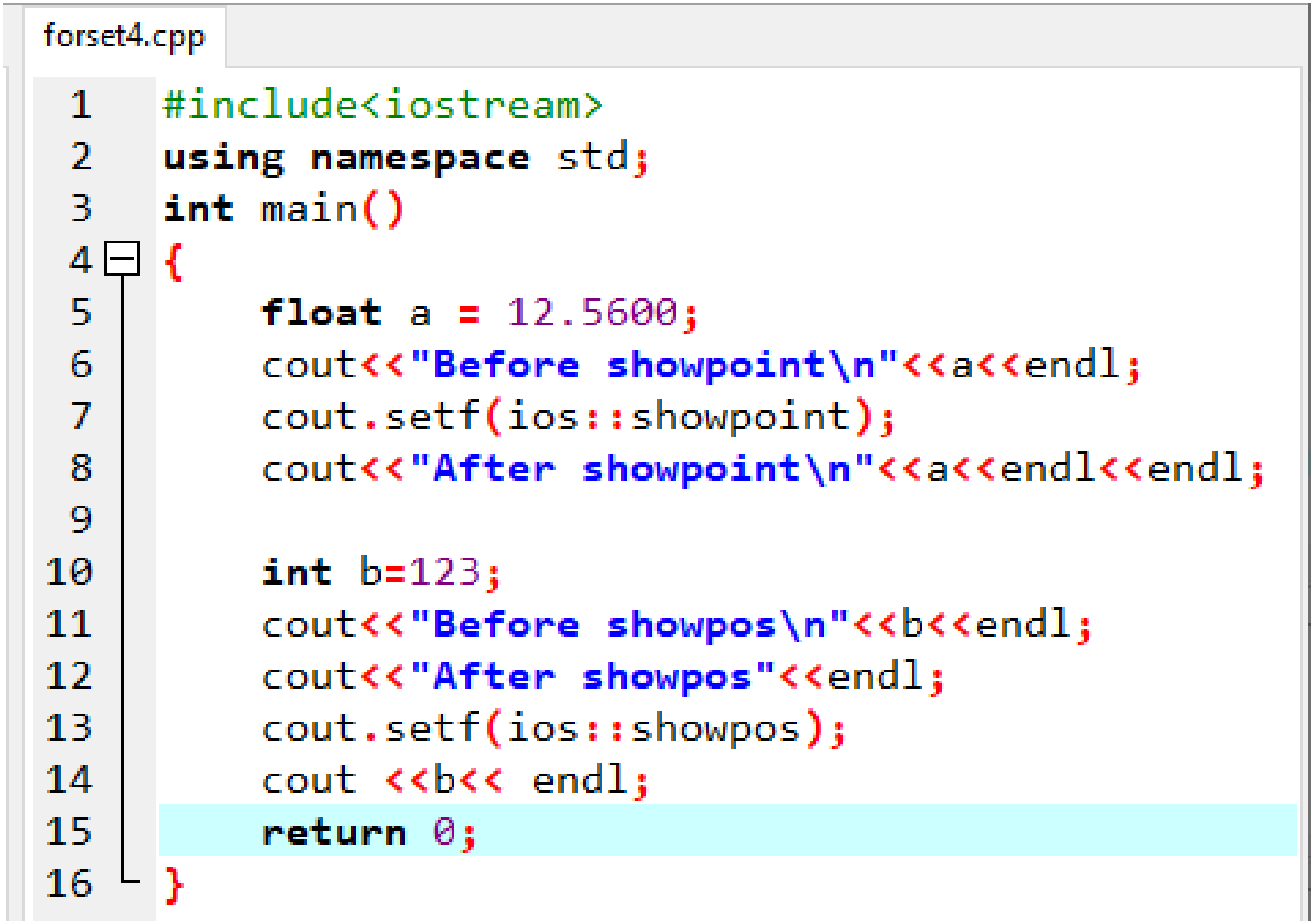
```
Hexadecimal Value of 100 = 64

Use of unsetf()
100
--------------------------------
Process exited after 0.01528 seconds with return value 0
Press any key to continue . . .
```

- **Displaying Trailing zeros and plus sign**

- **ios::showpoint**
  - Show trailing decimal point and zeros
- **ios::showpos**
  - Print + before positive numbers

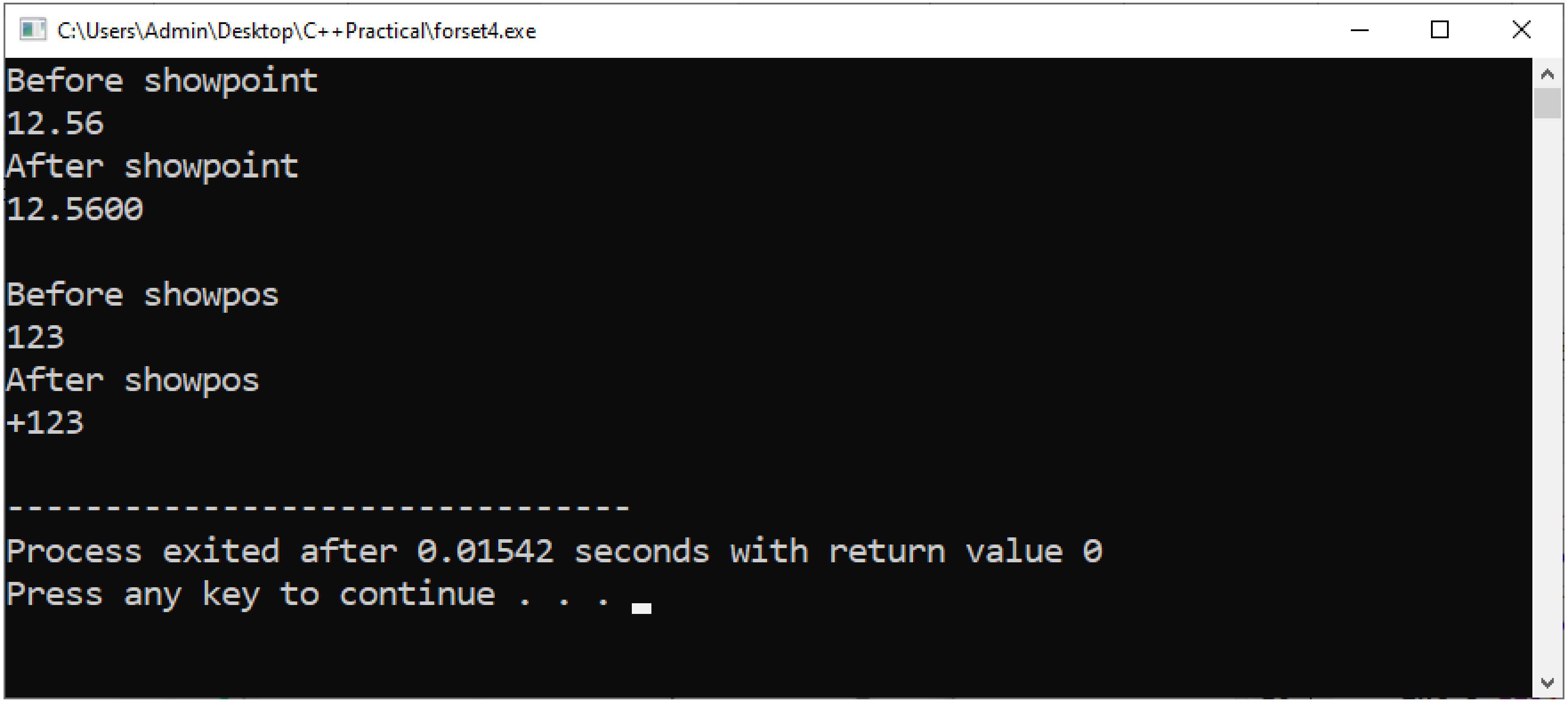
- The flags such as showpoint and showpos do not have any bit fields and therefore are used as **single arguments in setf().**

- E.g.



```cpp
#include<iostream>
using namespace std;
int main()
{
    float a = 12.5600;
    cout<<"Before showpoint\n"<<a<<endl;
    cout.setf(ios::showpoint);
    cout<<"After showpoint\n"<<a<<endl<<endl;

    int b=123;
    cout<<"Before showpos\n"<<b<<endl;
    cout<<"After showpos"<<endl;
    cout.setf(ios::showpos);
    cout <<b<< endl;
    return 0;
}
```

- Output:



```
Before showpoint
12.56
After showpoint
12.5600

Before showpos
123
After showpos
+123


--------------------------------
Process exited after 0.01542 seconds with return value 0
Press any key to continue . . .
```

Manipulators

- Manipulators **are functions** which are **used to manipulate the input/output formats.**
- Manipulators are special functions that are specifically designed to modify the working of a stream.
- All the **predefined manipulators** (set of functions) are **defined in the header file <u>iomanip</u>**
- **Some** of the **manipulators are more convenient** to **use than their counterparts in the class ios** as **<u>two or more manipulators can be used as a chain in one statement.</u>**

- This kind of **chaining is useful when** we want to **display several columns of output**.
- **Manipulators and the ios functions can be jointly use in the program**

- **NOTE**

some ios member function must used in their old format rather than the manipulators.

- The most commonly used manipulators are shown below:

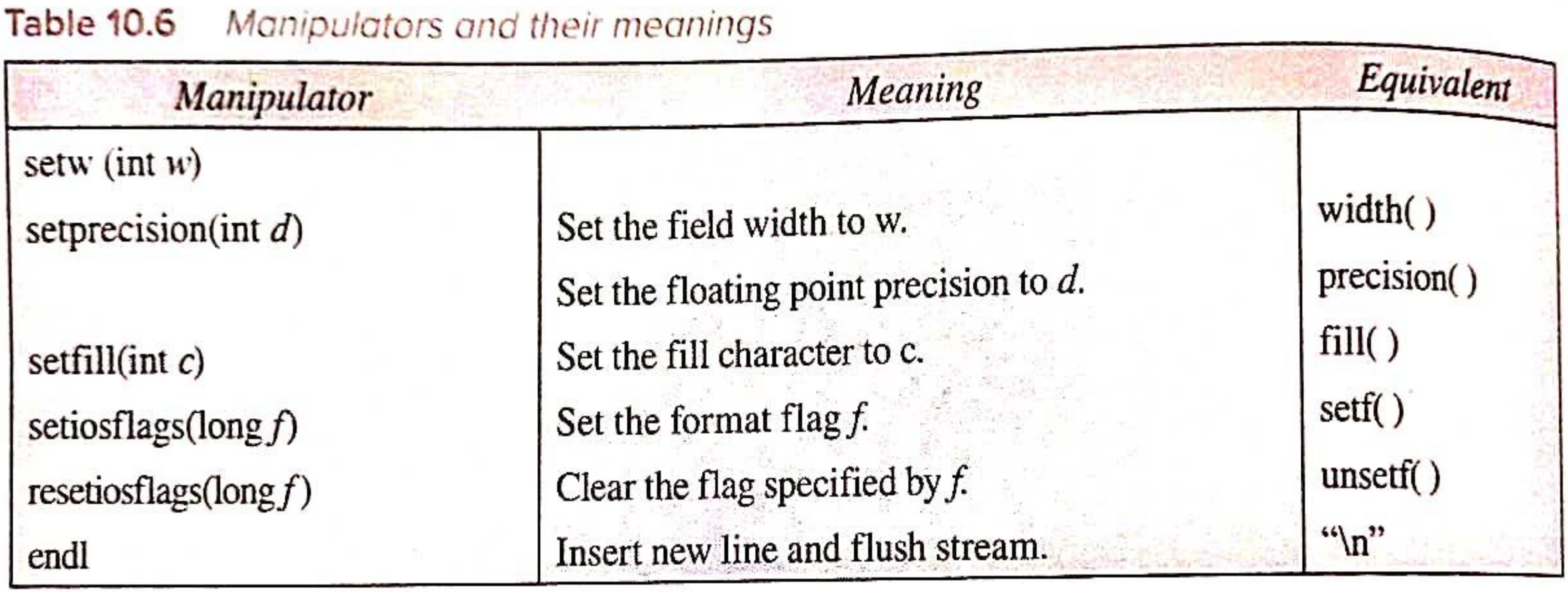
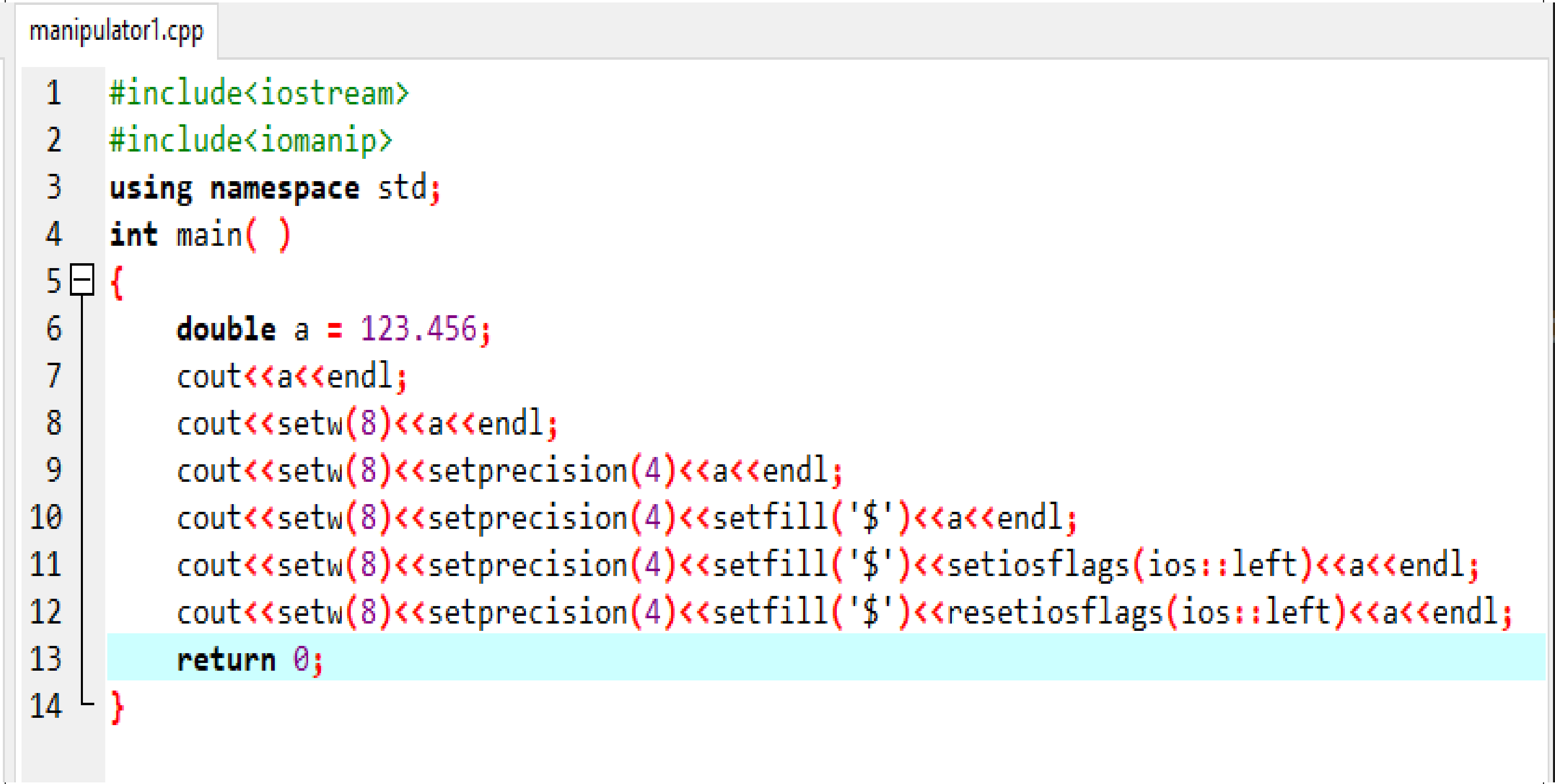


Table 10.6 *Manipulators and their meanings*

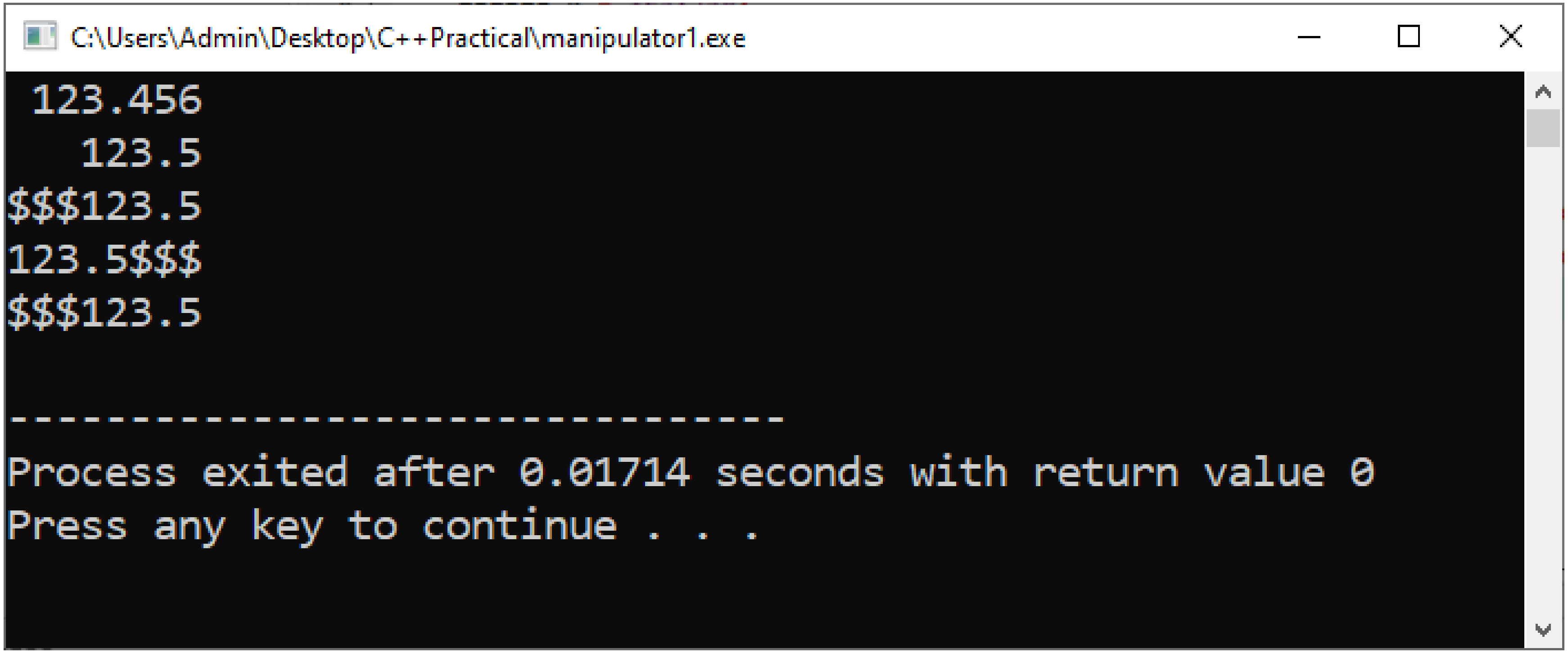
Manipulator	*Meaning*	*Equivalent*
setw (int *w*)	Set the field width to w.	width( )
setprecision(int *d*)	Set the floating point precision to *d*.	precision( )
setfill(int *c*)	Set the fill character to c.	fill( )
setiosflags(long *f*)	Set the format flag *f*.	setf( )
resetiosflags(long *f*)	Clear the flag specified by *f*.	unsetf( )
endl	Insert new line and flush stream.	“\n”

- E.g.



```
#include<iostream>
#include<iomanip>
using namespace std;
int main( )
{
    double a = 123.456;
    cout<<a<<endl;
    cout<<setw(8)<<a<<endl;
    cout<<setw(8)<<setprecision(4)<<a<<endl;
    cout<<setw(8)<<setprecision(4)<<setfill('$')<<a<<endl;
    cout<<setw(8)<<setprecision(4)<<setfill('$')<<setiosflags(ios::left)<<a<<endl;
    cout<<setw(8)<<setprecision(4)<<setfill('$')<<resetiosflags(ios::left)<<a<<endl;
    return 0;
}
```

- Output:



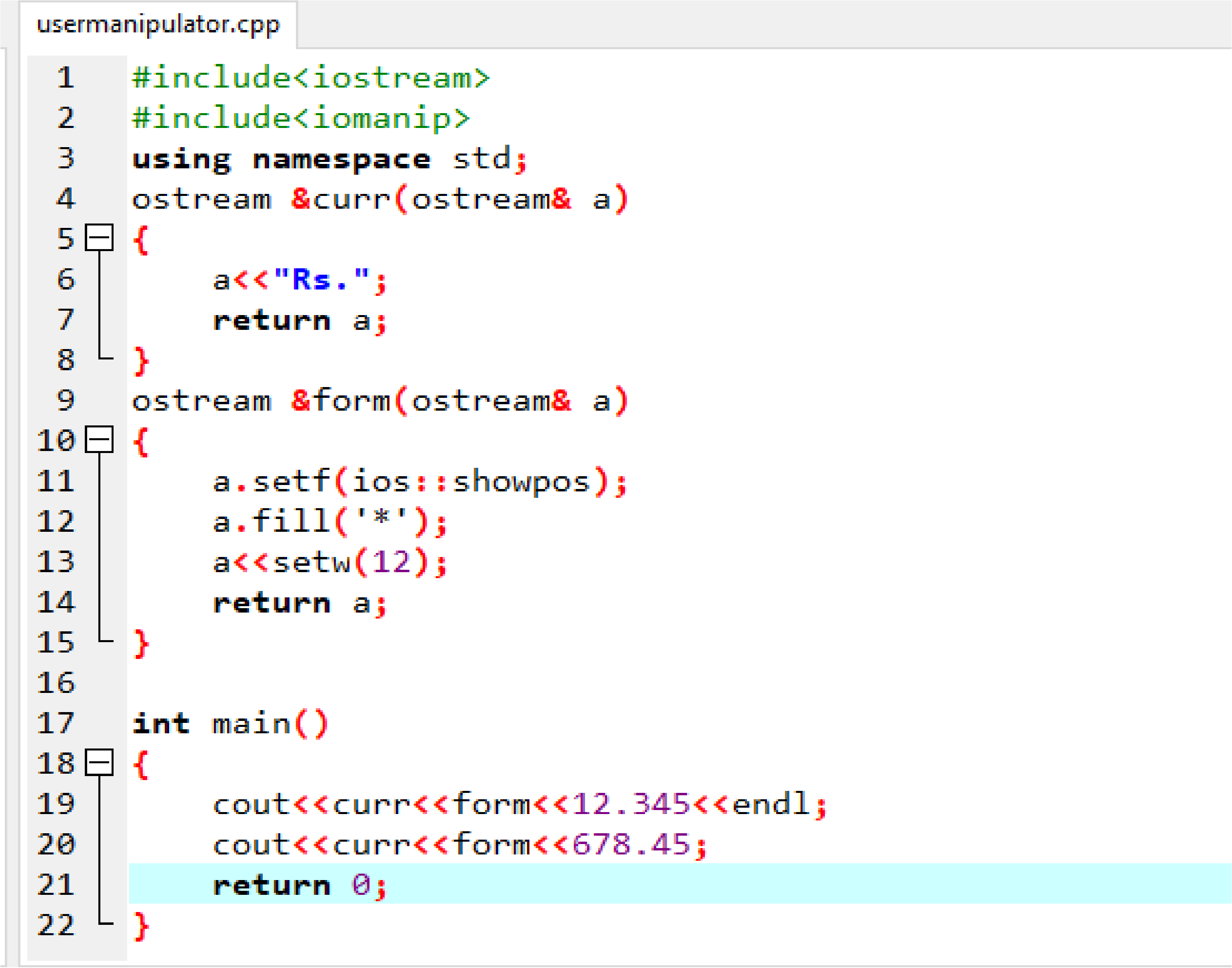
```
 123.456
   123.5
$$$123.5
123.5$$$
$$$123.5


--------------------------------
Process exited after 0.01714 seconds with return value 0
Press any key to continue . . .
```

User Defined Manipulators

- Apart from using the built-in manipulators of C++, the user defined manipulators can be created.
- In this, the **programmer can design their own manipulators based on their needs and requirements**
- **Syntax**

```
ostream & mipulator_name(ostream& output_name)
{
//code to be executed
return output_name;
}
```

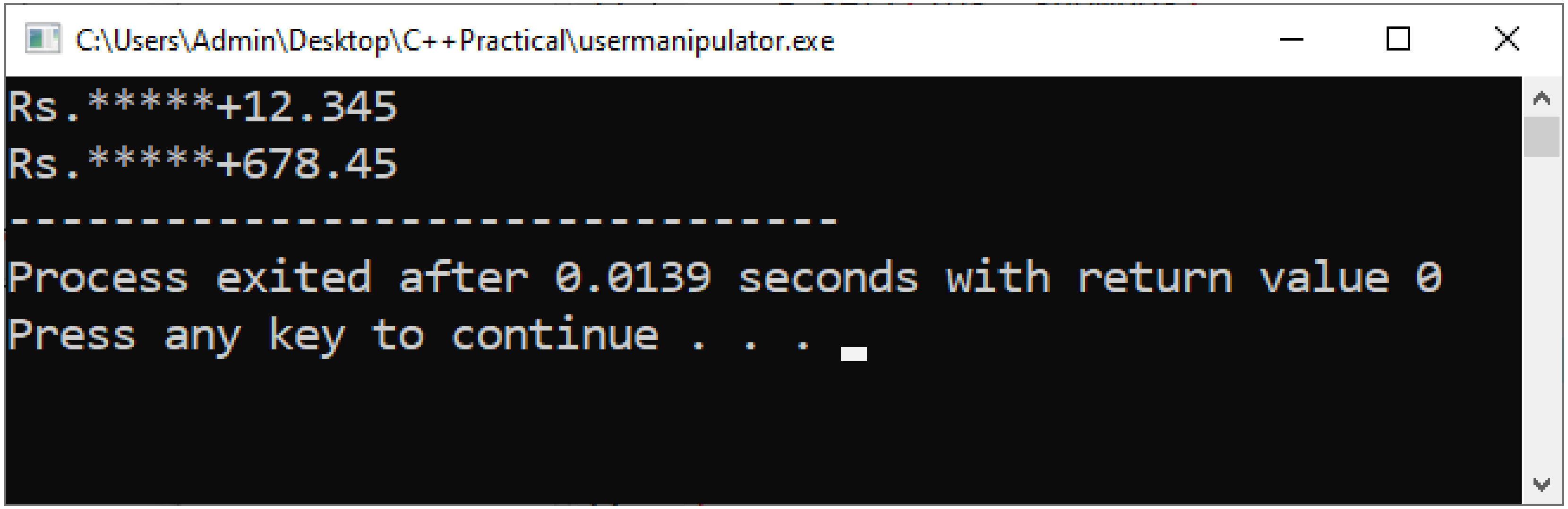
- E.g.



usermanipulator.cpp

```cpp
#include<iostream>
#include<iomanip>
using namespace std;
ostream &curr(ostream& a)
{
    a<<"Rs.";
    return a;
}
ostream &form(ostream& a)
{
    a.setf(ios::showpos);
    a.fill('*');
    a<<setw(12);
    return a;
}

int main()
{
    cout<<curr<<form<<12.345<<endl;
    cout<<curr<<form<<678.45;
    return 0;
}
```

- Output:



```
Rs.*****+12.345
Rs.*****+678.45
--------------------------------
Process exited after 0.0139 seconds with return value 0
Press any key to continue . . .
```