Object Oriented Programming (CS1143)

Week 9

Department of Computer Science

Capital University of Science and Technology (CUST)

Outline

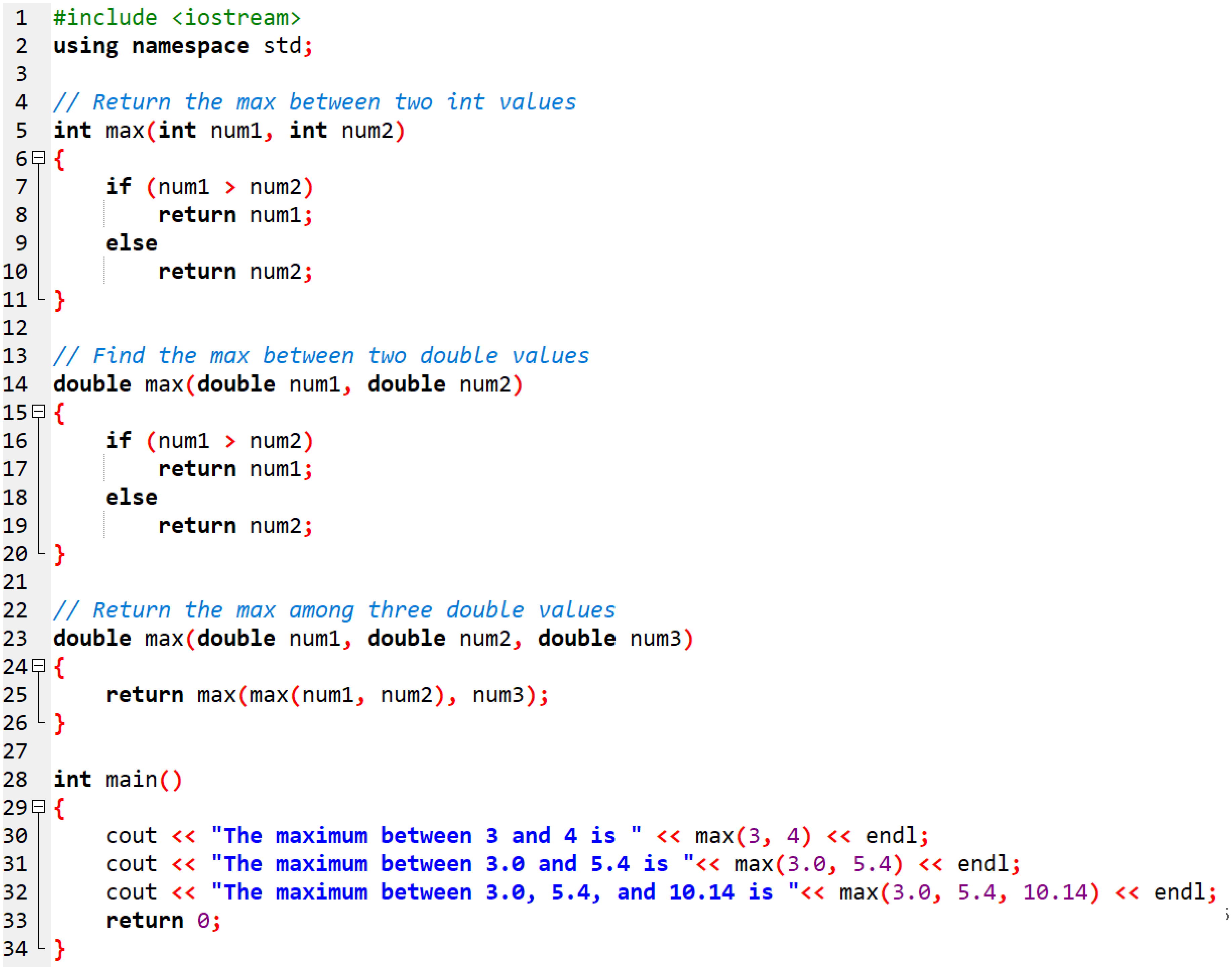
- Function Overloading
- Function Overriding
- Objects as argument to functions
- Pointer to Objects
- Introduction to Polymorphism

Outline

- Function Overloading
- Function Overriding
- Objects as argument to functions
- Pointer to Objects
- Introduction to Polymorphism

Function Overloading

- Overloading functions enables you to define the functions with the same name as long as their signatures are different.



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

// Return the max between two int values
int max(int num1, int num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}

// Find the max between two double values
double max(double num1, double num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}

// Return the max among three double values
double max(double num1, double num2, double num3)
{
    return max(max(num1, num2), num3);
}

int main()
{
    cout << "The maximum between 3 and 4 is " << max(3, 4) << endl;
    cout << "The maximum between 3.0 and 5.4 is "<< max(3.0, 5.4) << endl;
    cout << "The maximum between 3.0, 5.4, and 10.14 is "<< max(3.0, 5.4, 10.14) << endl;
    return 0;
}
```

Description

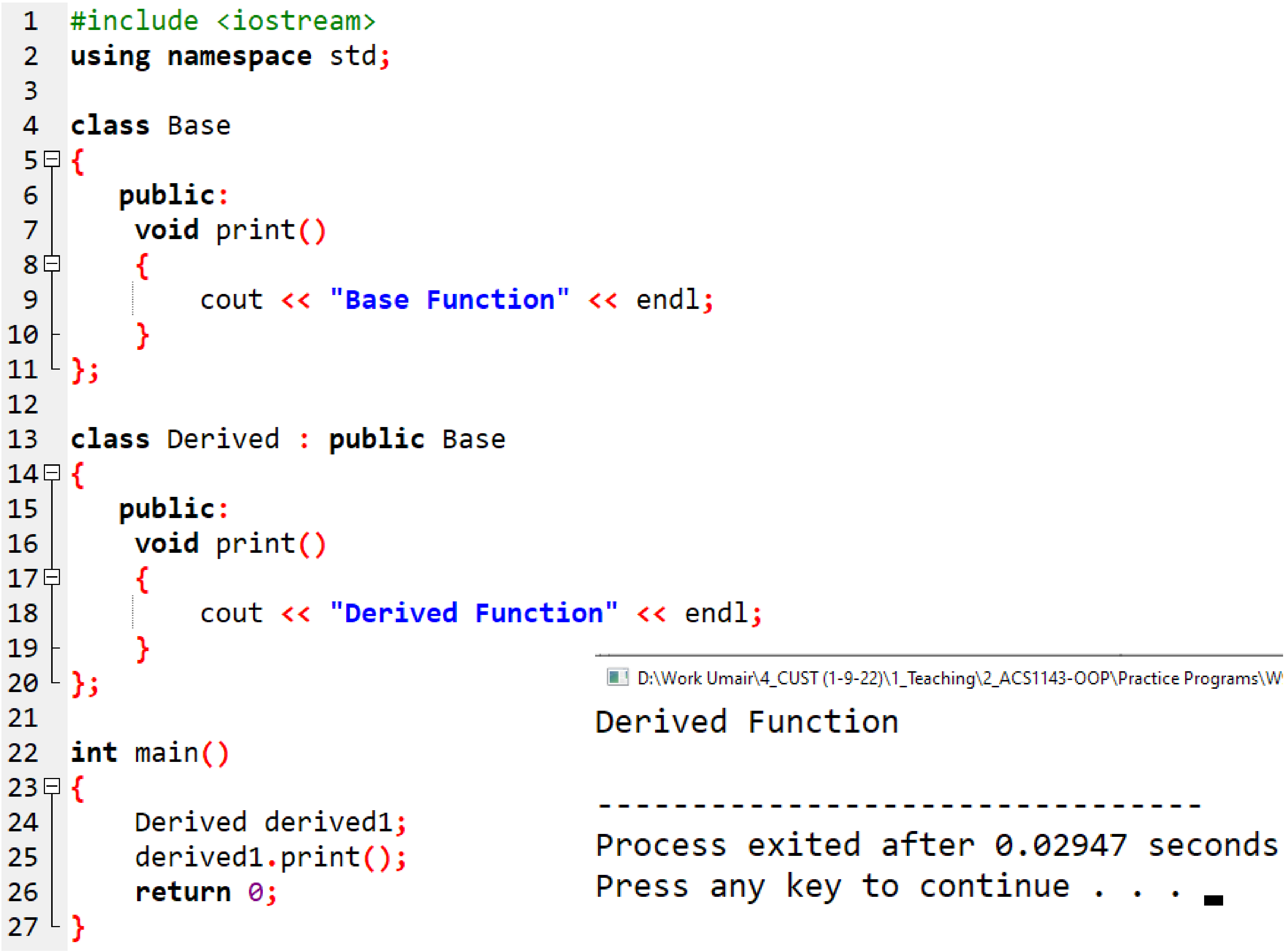
- If you call max with int parameters, the max function that expects int parameters will be invoked
- If you call max with double parameters, the max function that expects double parameters will be invoked.
- The C++ compiler determines which function is used based on the function signature.

Outline

- Function Overloading
- Function Overriding
- Objects as argument to functions
- Pointer to Objects
- Introduction to Polymorphism

Function Overriding

- Function overriding is a concept in object-oriented programming which allows a function within a derived class to override a function in its base class usually with a different implementation.
- A common use of function overriding is to provide a default implementation in the base class, and then overriding with a specific implementation in the derived class
- Redefining a Function



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

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

class Derived : public Base
{
   public:
    void print()
    {
        cout << "Derived Function" << endl;
    }
};

int main()
{
    Derived derived1;
    derived1.print();
    return 0;
}
```

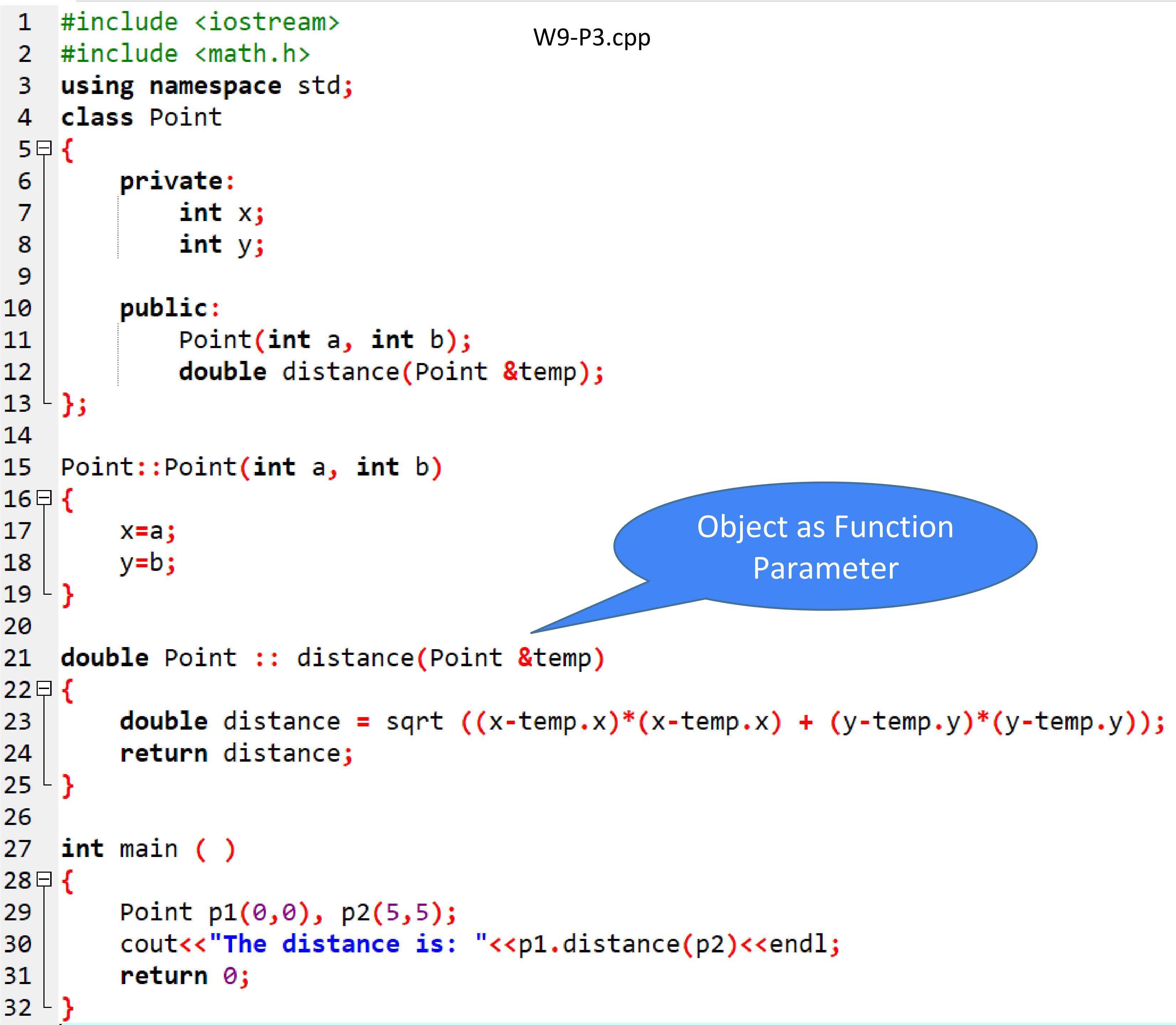
D:\Work Umair\4_CUST (1-9-22)\1_Teaching\2_ACS1143-OOP\Practice Programs\W

```
Derived Function

--------------------------------
Process exited after 0.02947 seconds
Press any key to continue . . .
```

Outline

- Function Overloading
- Function Overriding
- Objects as argument to functions
- Pointer to Objects
- Introduction to Polymorphism

W9-P3.cpp



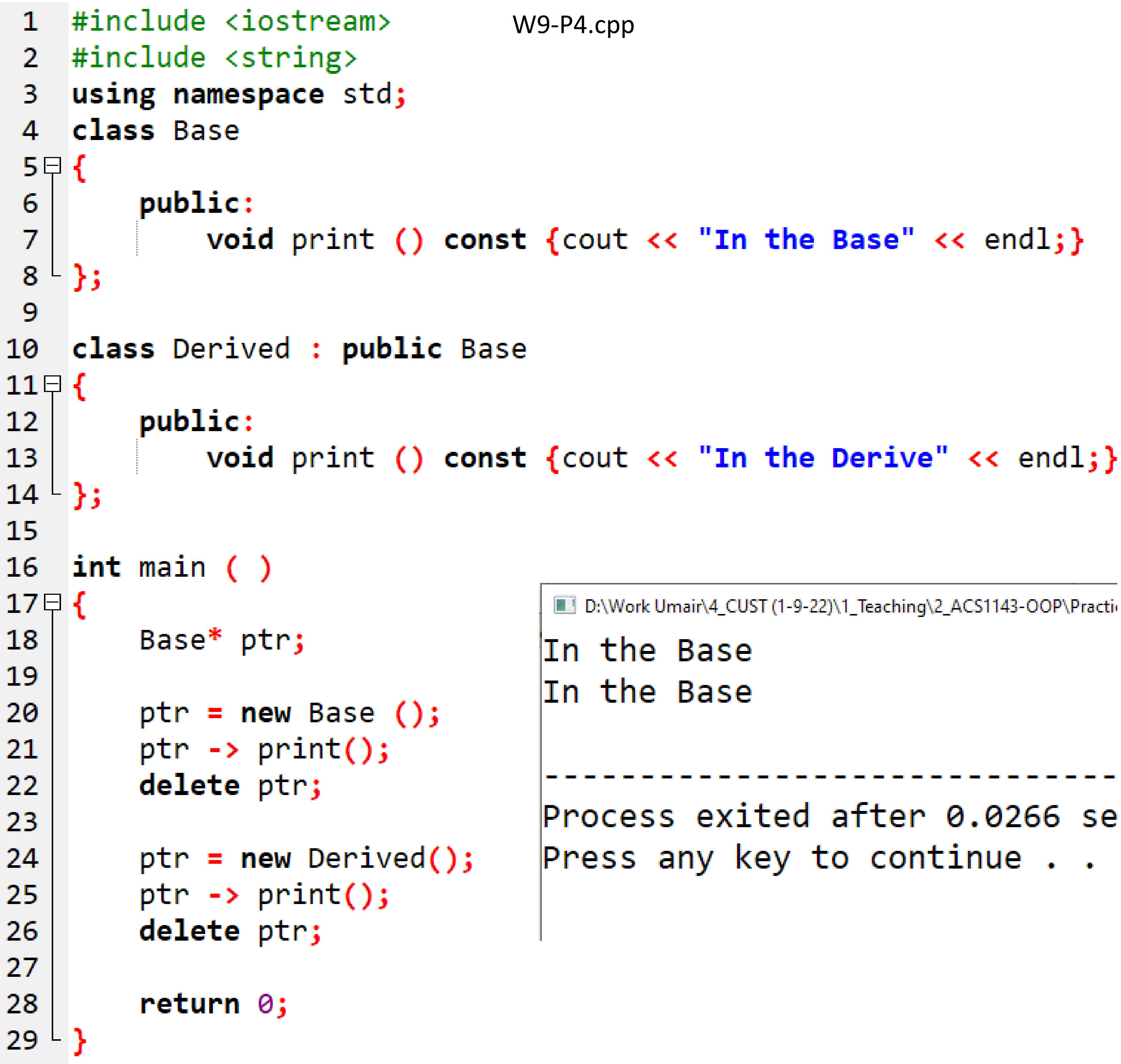
```cpp
#include <iostream>
#include <math.h>
using namespace std;
class Point
{
    private:
        int x;
        int y;

    public:
        Point(int a, int b);
        double distance(Point &temp);
};

Point::Point(int a, int b)
{
    x=a;
    y=b;
}

double Point :: distance(Point &temp)
{
    double distance = sqrt ((x-temp.x)*(x-temp.x) + (y-temp.y)*(y-temp.y));
    return distance;
}

int main ( )
{
    Point p1(0,0), p2(5,5);
    cout<<"The distance is: "<<p1.distance(p2)<<endl;
    return 0;
}
```

Object as Function Parameter

Outline

- Function Overloading
- Function Overriding
- Objects as argument to functions
- Pointer to Objects
- Introduction to Polymorphism

W9-P4.cpp



```cpp
#include <iostream>
#include <string>
using namespace std;
class Base
{
    public:
        void print () const {cout << "In the Base" << endl;}
};

class Derived : public Base
{
    public:
        void print () const {cout << "In the Derive" << endl;}
};

int main ( )
{
    Base* ptr;

    ptr = new Base ();
    ptr -> print();
    delete ptr;

    ptr = new Derived();
    ptr -> print();
    delete ptr;

    return 0;
}
```

D:\Work Umair\4_CUST (1-9-22)\1_Teaching\2_ACS1143-OOP\Practi

```
In the Base
In the Base

--------------------------------
Process exited after 0.0266 se
Press any key to continue . .
```

Description

- At line 20, ptr is pointing to an object of the Base class, and at line 21, we call the function defined in the Base class.
- At line 24, we make the same pointer point to an object of the Derived class, and at line 25, we tried to call the function defined in the Derived class

- The result shows that the function defined for the Base class is called both times.

Outline

- Function Overloading
- Function Overriding
- Objects as argument to functions
- Pointer to Objects
- Introduction to Polymorphism

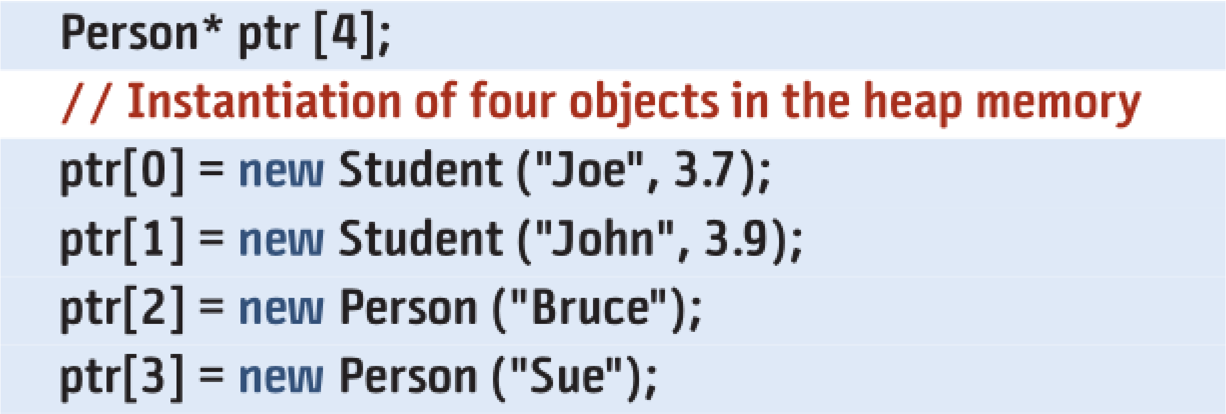
Why we need pointers to objects?

- We did not need to use pointers in the program on Slide 13.
- We can use base class’s object (say b1) and write b1.print( ) instead of ptr −> print()
- We can use derived class’s object (say d1) and write d1.print( ) instead of ptr −> print().
- However, the program shows the idea where we can use only one pointer that can point to different objects.

- Assume we need to have an array of objects.
- We know that all elements of an array must be of the same type; this means we cannot use an array of objects if the objects are of different types.
- **However, we can use an array of pointers, in which each pointer can point to an object of the base class.**

Example

- Assume that we have a base class “Person” and a class “Student” derived from it.
- We can create an array of pointers where each pointer can point to an object of the base class (Person).
- We can store objects of both Person and Student class in this array.



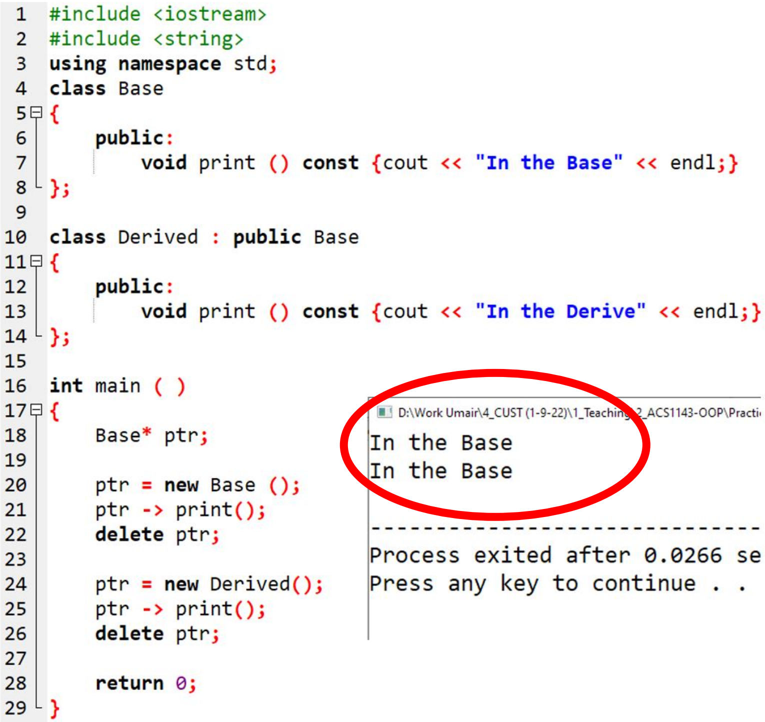
```
Person* ptr [4];
// Instantiation of four objects in the heap memory
ptr[0] = new Student ("Joe", 3.7);
ptr[1] = new Student ("John", 3.9);
ptr[2] = new Person ("Bruce");
ptr[3] = new Person ("Sue");
```

Polymorphism

- A function can be implemented in several classes along the inheritance chain (Function Overriding).
- **There must be a way for the system to decide which function is invoked (i.e. from which class) at runtime based on the actual type of the object stored in the pointer.**
- This is commonly known as polymorphism (from a Greek word meaning “many forms”).

Enabling Polymorphism

- In the program on Slide 13 (shown here again), in both cases the function of the base class was called.
- We can solve this problem if we declare the function in the base class as “virtual”.
- This is done using the keyword “virtual”
- Now the appropriate function will be called based on the type of object.

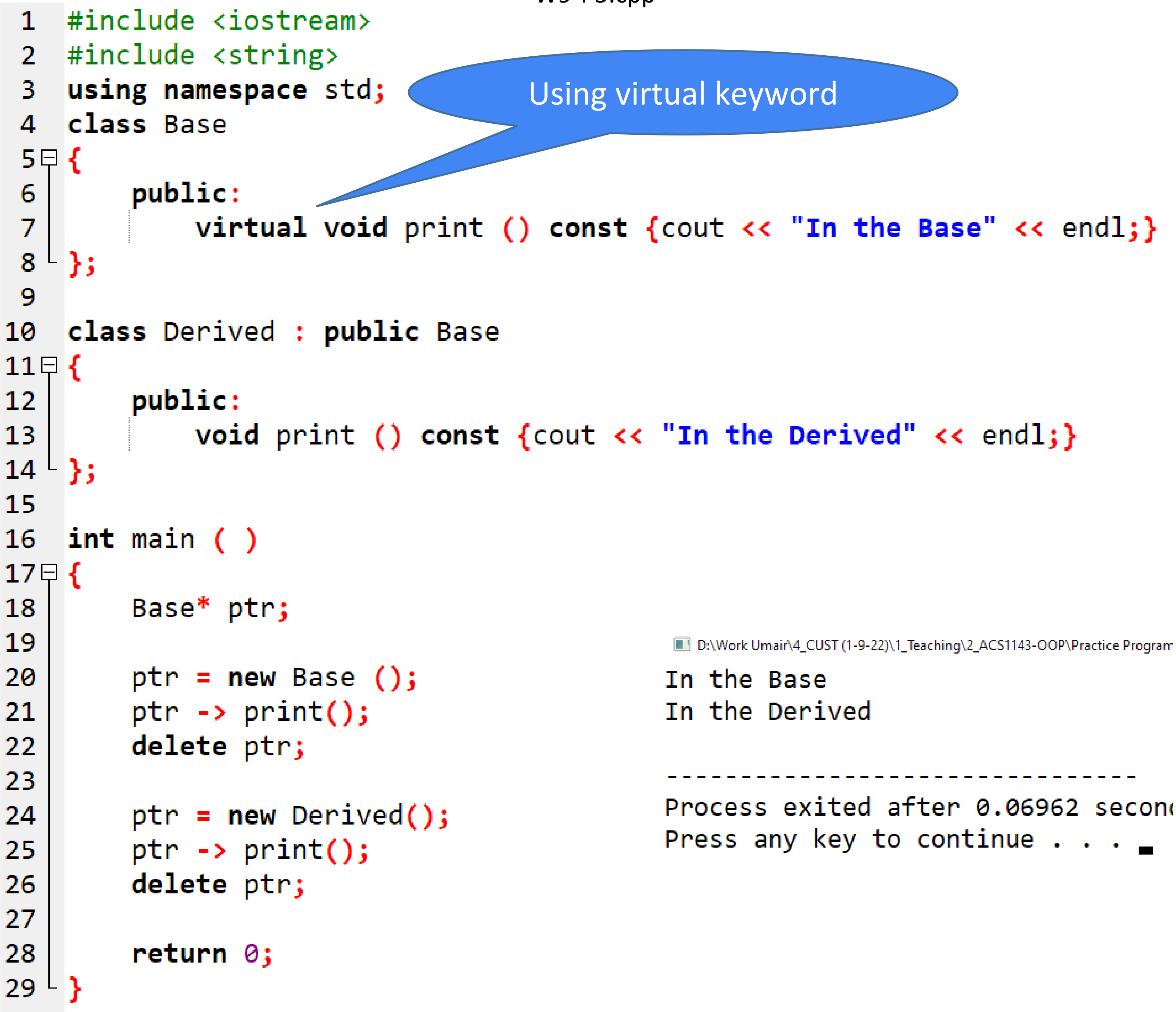


```cpp
#include <iostream>
#include <string>
using namespace std;
class Base
{
    public:
        void print () const {cout << "In the Base" << endl;}
};

class Derived : public Base
{
    public:
        void print () const {cout << "In the Derive" << endl;}
};

int main ( )
{
    Base* ptr;

    ptr = new Base ();
    ptr -> print();
    delete ptr;

    ptr = new Derived();
    ptr -> print();
    delete ptr;

    return 0;
}
```

D:\Work Umair\4_CUST (1-9-22)\1_Teaching\2_ACS1143-OOP\Practi

```
In the Base
In the Base

--------------------------------
Process exited after 0.0266 se
Press any key to continue . .
```

W9-P5.cpp



```cpp
#include <iostream>
#include <string>
using namespace std;
class Base
{
    public:
        virtual void print () const {cout << "In the Base" << endl;}
};

class Derived : public Base
{
    public:
        void print () const {cout << "In the Derived" << endl;}
};

int main ( )
{
    Base* ptr;

    ptr = new Base ();
    ptr -> print();
    delete ptr;

    ptr = new Derived();
    ptr -> print();
    delete ptr;

    return 0;
}
```

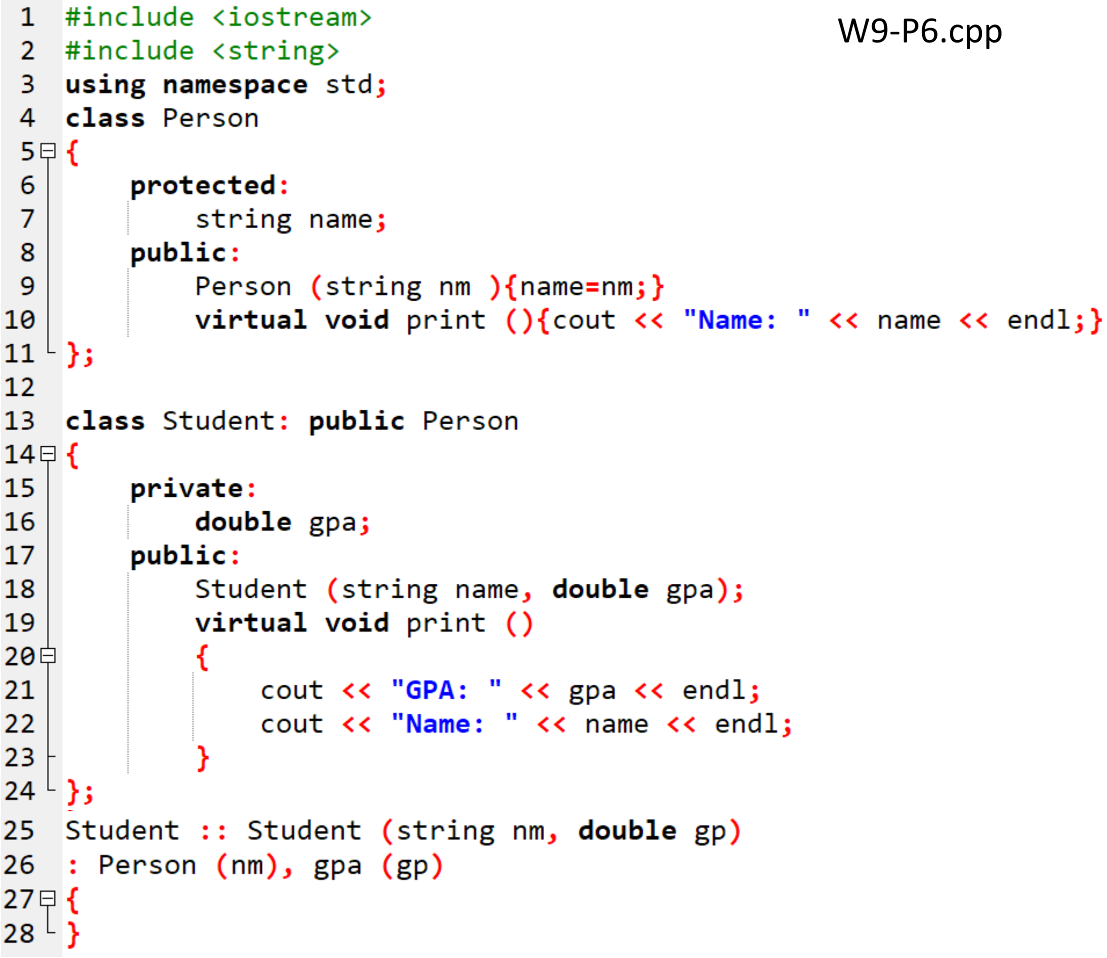
Using virtual keyword

D:\Work Umair\4_CUST (1-9-22)\1_Teaching\2_ACS1143-OOP\Practice Program

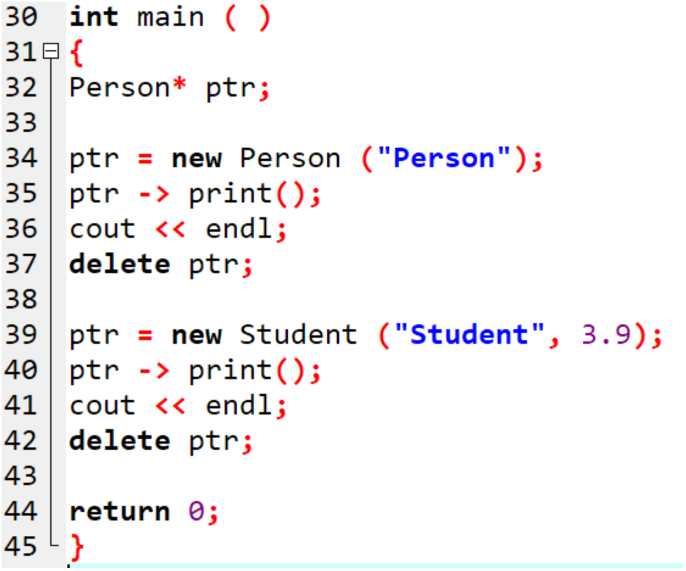
```
In the Base
In the Derived

--------------------------------
Process exited after 0.06962 secon
Press any key to continue . . .
```

Another Example

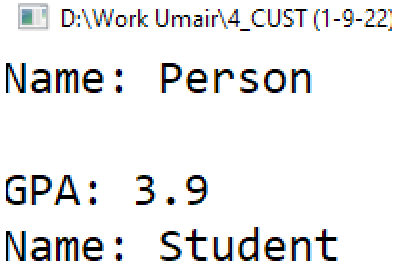
W9-P6.cpp



```cpp
#include <iostream>
#include <string>
using namespace std;
class Person
{
    protected:
        string name;
    public:
        Person (string nm ){name=nm;}
        virtual void print (){cout << "Name: " << name << endl;}
};

class Student: public Person
{
    private:
        double gpa;
    public:
        Student (string name, double gpa);
        virtual void print ()
        {
            cout << "GPA: " << gpa << endl;
            cout << "Name: " << name << endl;
        }
};
Student :: Student (string nm, double gp)
: Person (nm), gpa (gp)
{
}
```



```cpp
int main ( )
{
Person* ptr;

ptr = new Person ("Person");
ptr -> print();
cout << endl;
delete ptr;

ptr = new Student ("Student", 3.9);
ptr -> print();
cout << endl;
delete ptr;

return 0;
}
```



```
Name:  Person

GPA:  3.9
Name:  Student
```

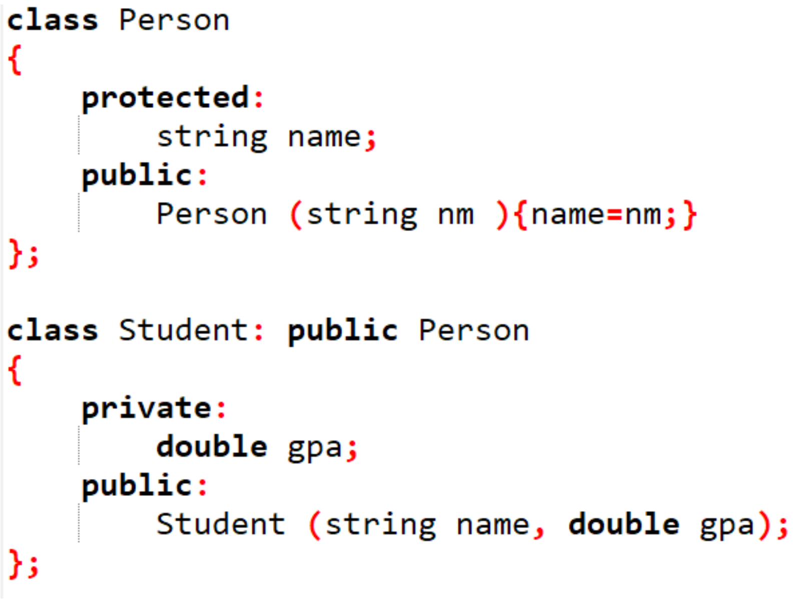
Constructors and Destructors

Constructors and Destructors

- Constructors cannot be virtual because although constructors are member functions, the names of the constructors are different for the base and derived classes (different signatures).
- Although the names of the destructors differ in the base and derived classes, the destructors are not normally called by their name.
- When there is a virtual member function anywhere in the design, we should also make the destructors virtual to avoid memory leaks.
- To understand the situation, we discuss two cases:
  - (1) when we are not using polymorphism,
  - (2) when we are using polymorphism.

Case 1: No Polymorphism

- Assume that we create a Person class and a Student class.
- The Person class has a name data member of type **string** in which the characters are created in the heap.
  - **For data members of type string, even though the object is in the stack, the characters representing the string are allocated in the heap**
- The Student class inherits name from the Person class, but it also adds another data member, gpa.
  - Since the derived class inherits the string data member, the derived class also has a data member allocated in heap memory



```
class Person
{
    protected:
        string name;
    public:
        Person (string nm ){name=nm;}
};

class Student: public Person
{
    private:
        double gpa;
    public:
        Student (string name, double gpa);
};
```

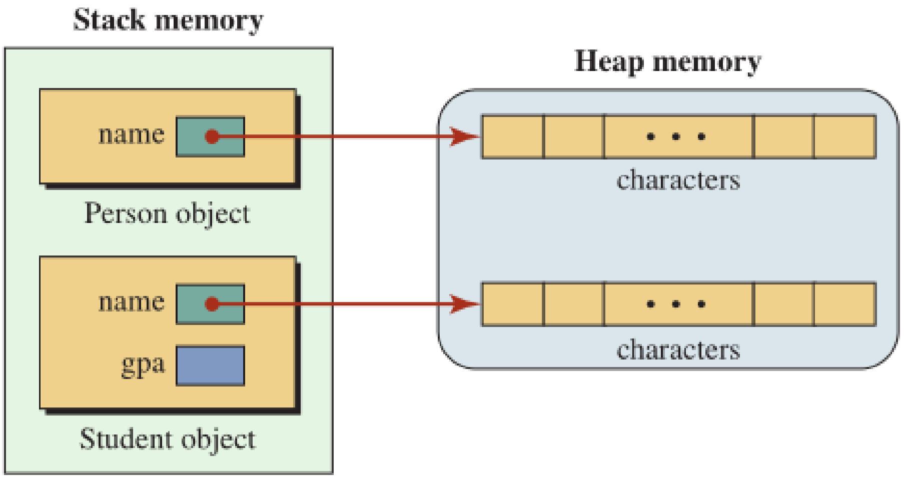
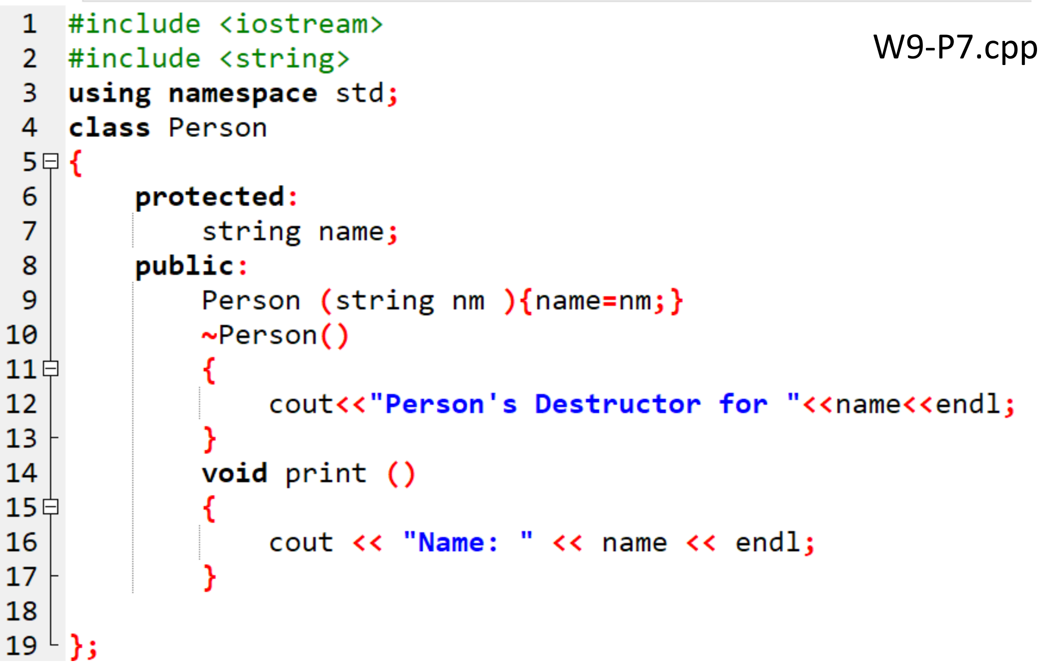


Figure 12.3 Two objects in a program when polymorphism is not used

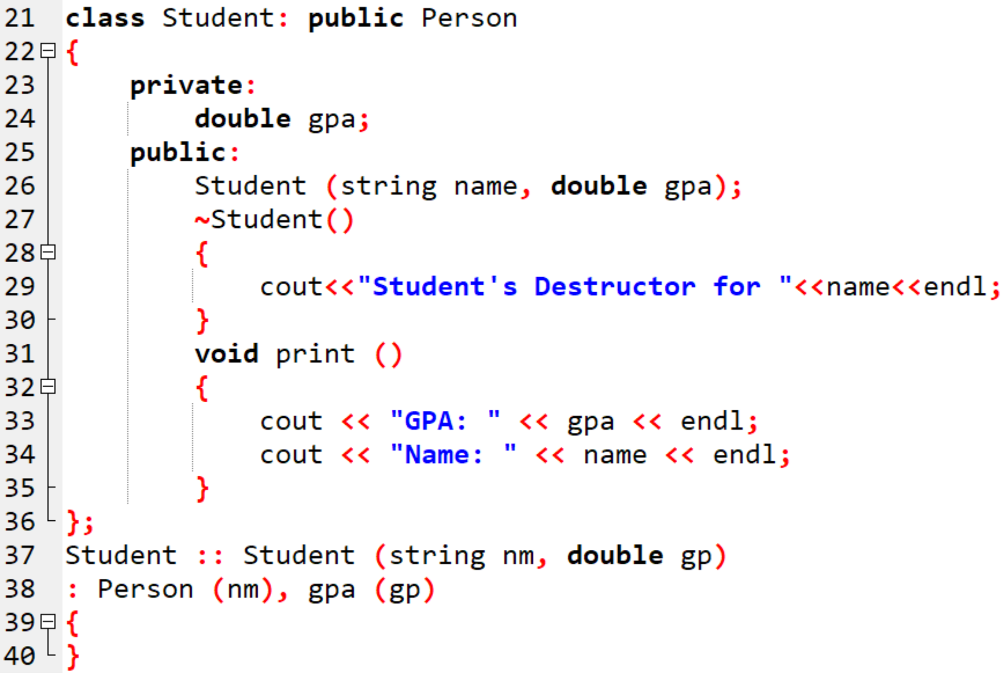
Case 1: No Polymorphism Contd..

- We cannot have a memory leak in this situation.
- When the program terminates, **the destructors for Person class and Student class are called**, which automatically call the destructors of the string class, which delete the allocated memory in the heap.

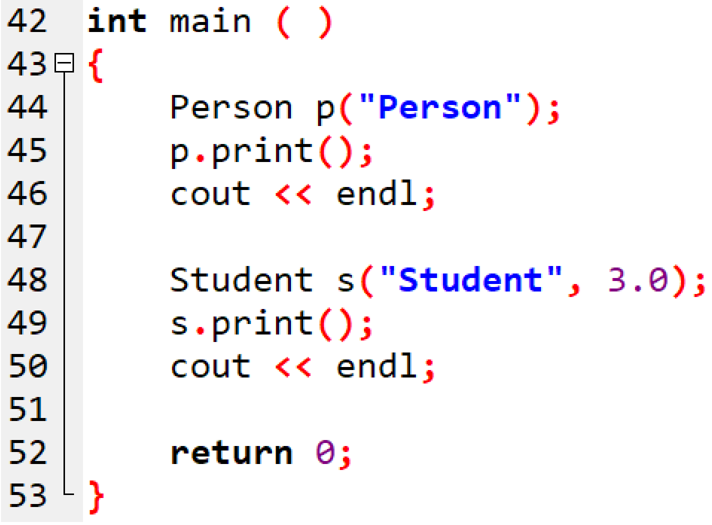
W9-P7.cpp



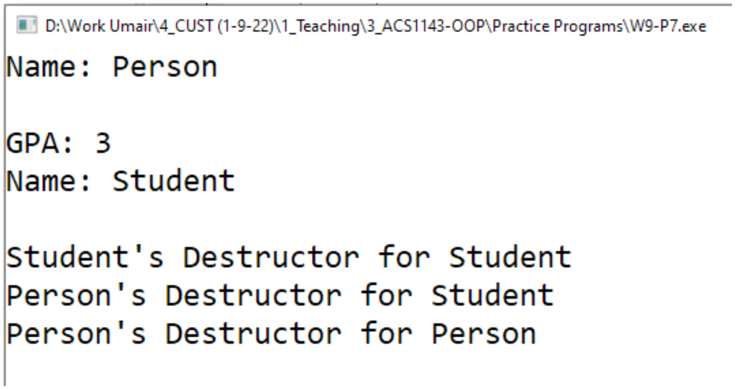
```cpp
#include <iostream>
#include <string>
using namespace std;
class Person
{
	protected:
		string name;
	public:
		Person (string nm ){name=nm;}
		~Person()
		{
			cout<<"Person's Destructor for "<<name<<endl;
		}
		void print ()
		{
			cout << "Name: " << name << endl;
		}

};
```



```cpp
class Student: public Person
{
	private:
		double gpa;
	public:
		Student (string name, double gpa);
		~Student()
		{
			cout<<"Student's Destructor for "<<name<<endl;
		}
		void print ()
		{
			cout << "GPA: " << gpa << endl;
			cout << "Name: " << name << endl;
		}
};
Student :: Student (string nm, double gp)
: Person (nm), gpa (gp)
{
}
```



```cpp
int main ( )
{
	Person p("Person");
	p.print();
	cout << endl;

	Student s("Student", 3.0);
	s.print();
	cout << endl;

	return 0;
}
```



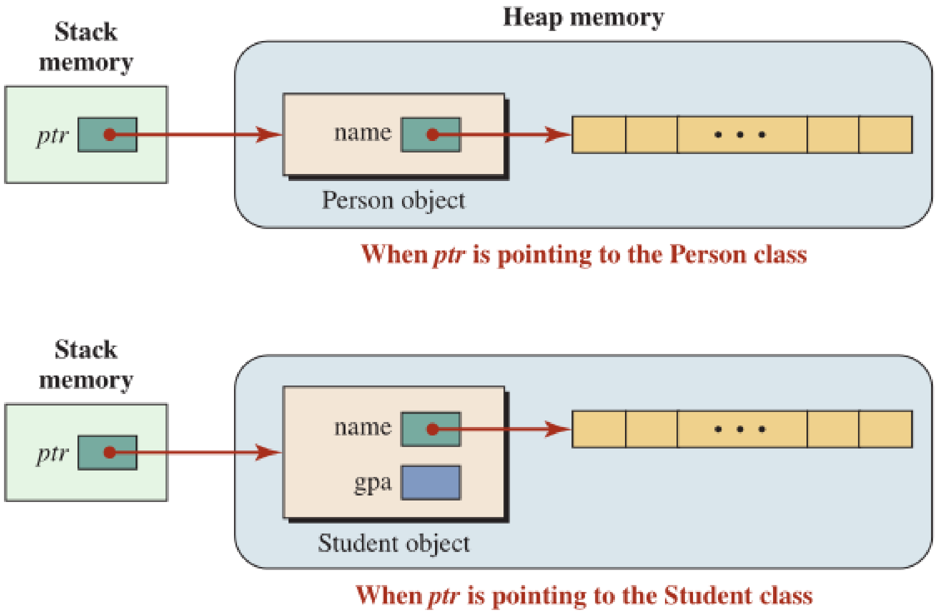
D:\Work Umair\4_CUST (1-9-22)\1_Teaching\3_ACS1143-OOP\Practice Programs\W9-P7.exe

```
Name: Person

GPA: 3
Name: Student

Student's Destructor for Student
Person's Destructor for Student
Person's Destructor for Person
```

No Polymorphism. The destructors for Person class and Student class have been called

Case 2: Polymorphism

- With Polymorphism, the situation is different.
- The Person object and the Student object are created in the heap. The string objects are also created in the heap.



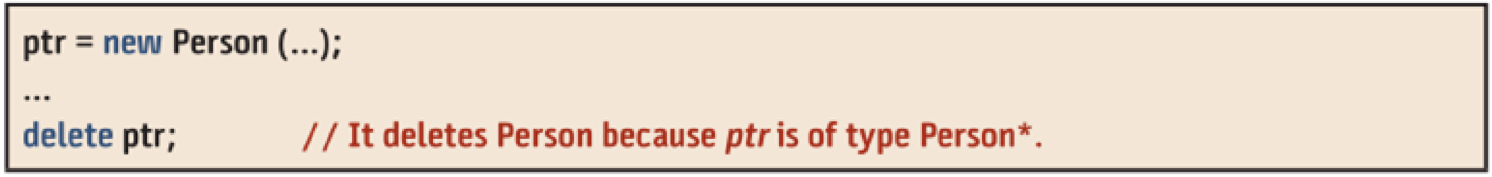
When *ptr* is pointing to the Person class

When *ptr* is pointing to the Student class

Figure 12.4 Two objects in a program when polymorphism is used

Case 2: Polymorphism – No Memory Leak

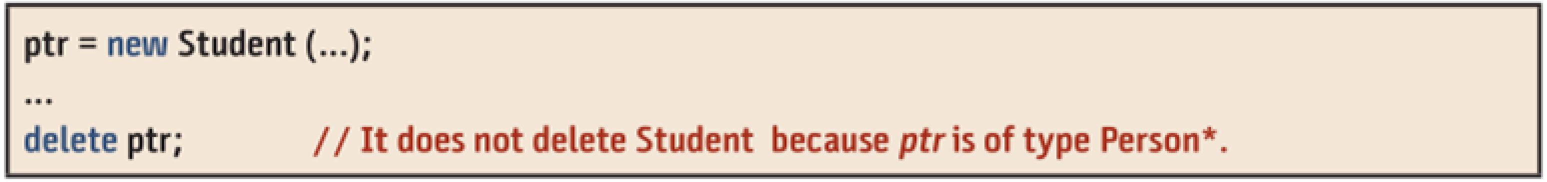
- We apply the delete operator to the polymorphic variable, ptr, in stack memory to delete the objects in heap memory.



```
ptr = new Person (...);
...
delete ptr;          // It deletes Person because ptr is of type Person*.
```

- In the above scenario, the pointer **ptr** is a pointer to Person type and the delete operator can delete the Person object.
- When the Person object is deleted, its destructor is called, which in turn calls the destructor of the string class. The characters created in the heap are de-allocated.
- **There is no memory leak**.

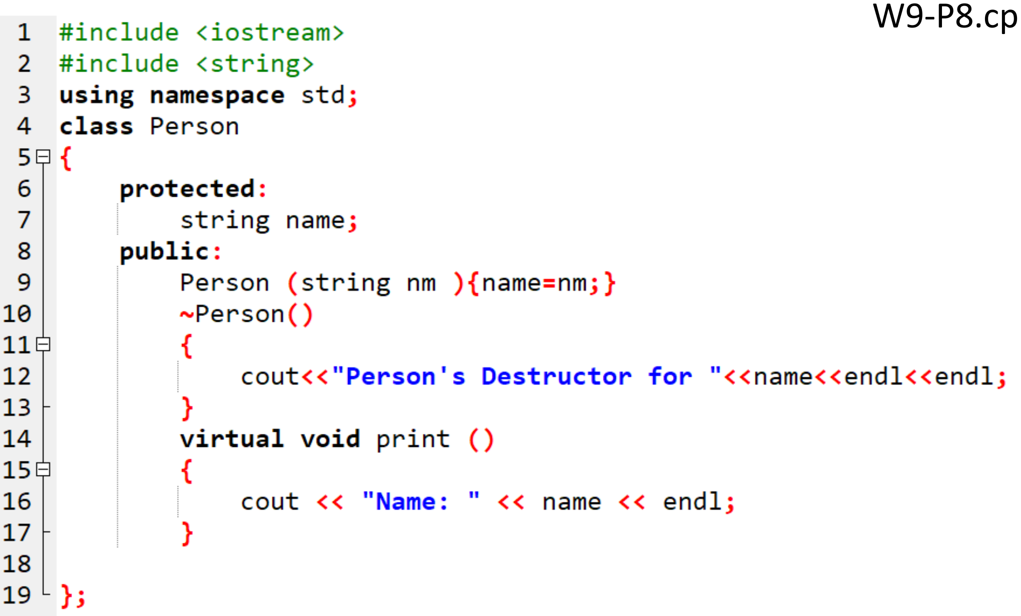
Case 2: Polymorphism – Memory Leak



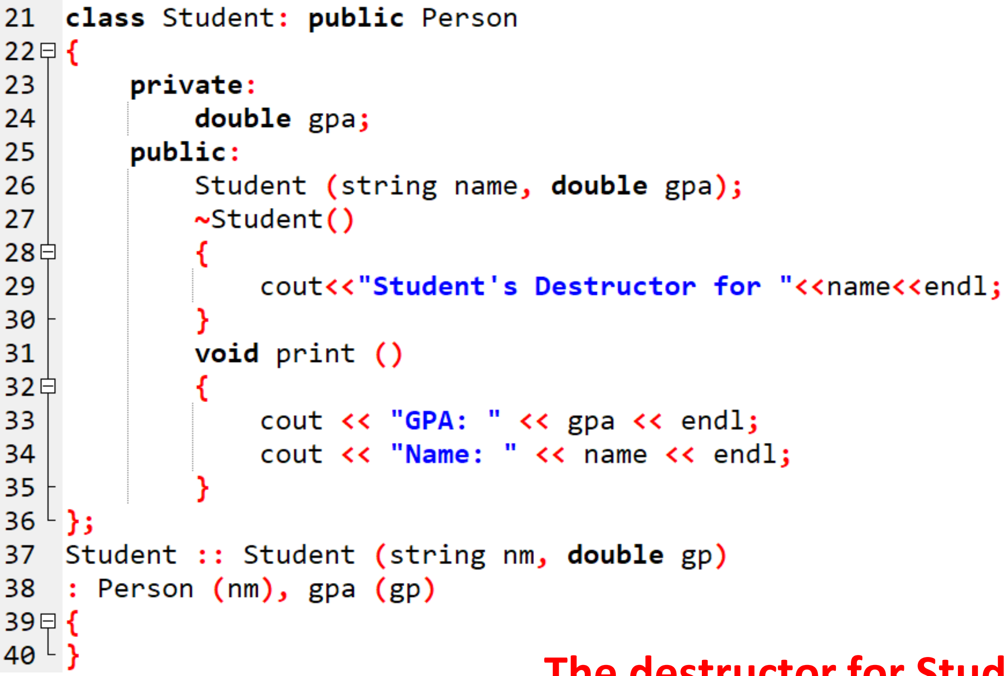
```
ptr = new Student (...);
...
delete ptr;          // It does not delete Student  because ptr is of type Person*.
```

- In the above scenario, the pointer ptr is still a pointer to Person type, which means it can delete an object of a Person class (which does not exist and nothing happens), but it cannot delete the object of the Student class.
- **When the object of the Student class is not deleted, its destructor is not called**, which means that the destructor of the string class is also not called, which means the characters in the heap are not de-allocated.
- **We have memory leak**.

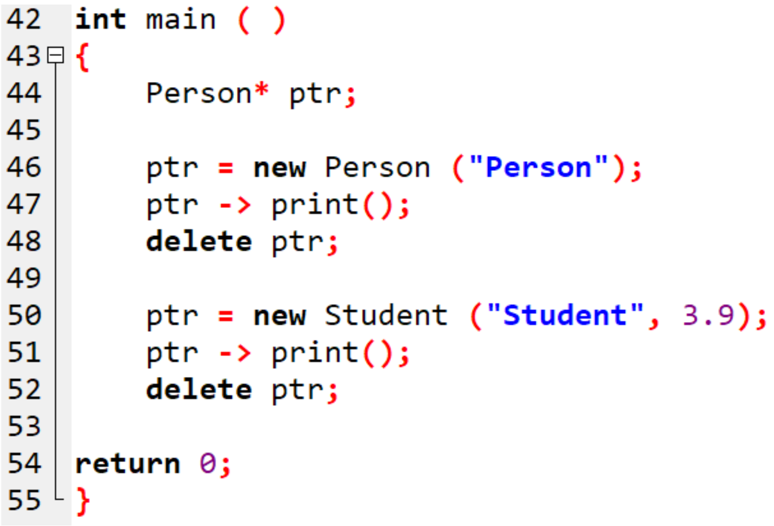
W9-P8.cpp



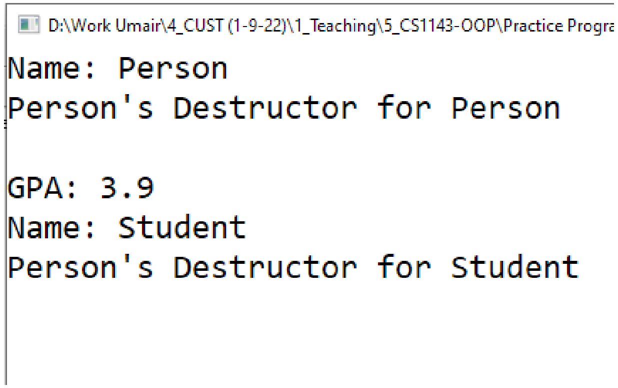
```cpp
#include <iostream>
#include <string>
using namespace std;
class Person
{
    protected:
        string name;
    public:
        Person (string nm ){name=nm;}
        ~Person()
        {
            cout<<"Person's Destructor for "<<name<<endl<<endl;
        }
        virtual void print ()
        {
            cout << "Name: " << name << endl;
        }

};
```



```cpp
class Student: public Person
{
    private:
        double gpa;
    public:
        Student (string name, double gpa);
        ~Student()
        {
            cout<<"Student's Destructor for "<<name<<endl;
        }
        void print ()
        {
            cout << "GPA: " << gpa << endl;
            cout << "Name: " << name << endl;
        }
};
Student :: Student (string nm, double gp)
: Person (nm), gpa (gp)
{
}
```



```cpp
int main ( )
{
    Person* ptr;

    ptr = new Person ("Person");
    ptr -> print();
    delete ptr;

    ptr = new Student ("Student", 3.9);
    ptr -> print();
    delete ptr;

return 0;
}
```



D:\Work Umair\4_CUST (1-9-22)\1_Teaching\5_CS1143-OOP\Practice Progra

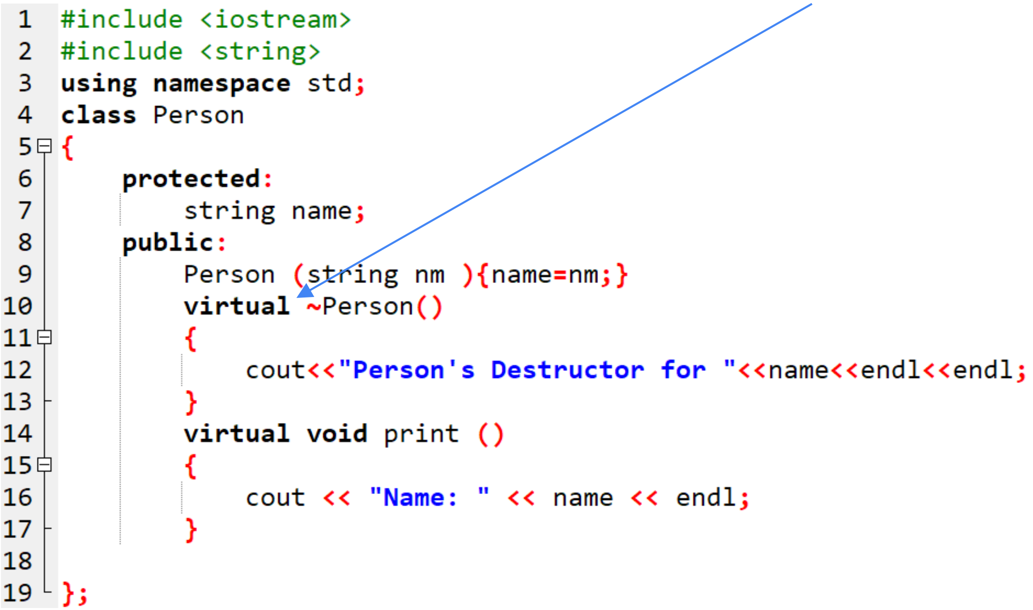
```
Name: Person
Person's Destructor for Person

GPA: 3.9
Name: Student
Person's Destructor for Student
```

The destructor for Student class has not been called

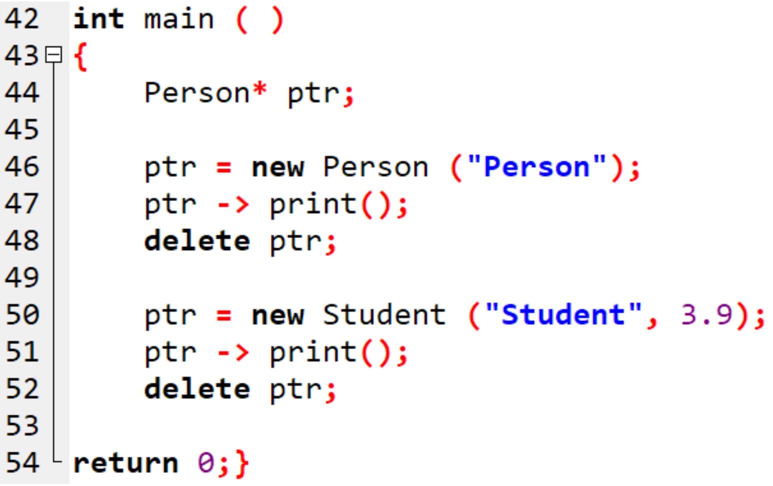
Solution

- The solution is to **make the destructor of the base class virtual**, which automatically makes the destructor of the derived class virtual.
- In this case, the system allows two different member functions with different names to be virtual

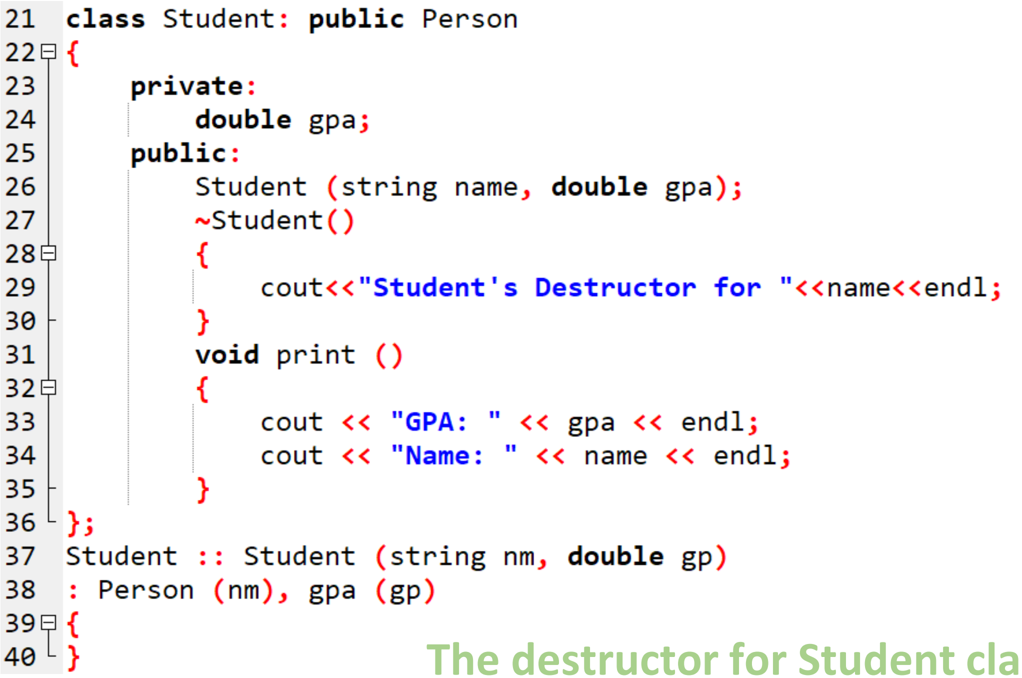
W9-P9.cpp



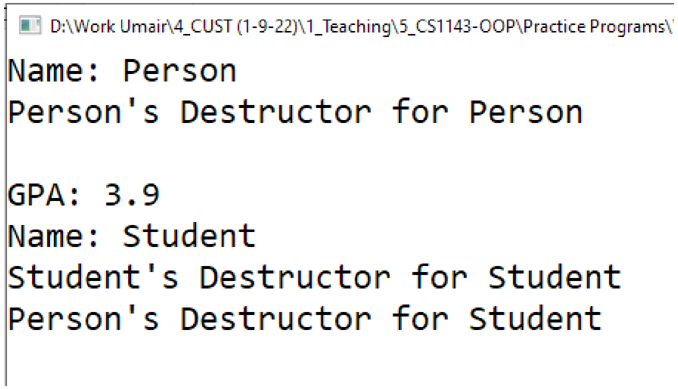
```cpp
#include <iostream>
#include <string>
using namespace std;
class Person
{
    protected:
        string name;
    public:
        Person (string nm ){name=nm;}
        virtual ~Person()
        {
            cout<<"Person's Destructor for "<<name<<endl<<endl;
        }
        virtual void print ()
        {
            cout << "Name: " << name << endl;
        }

};
```



```cpp
int main ( )
{
    Person* ptr;

    ptr = new Person ("Person");
    ptr -> print();
    delete ptr;

    ptr = new Student ("Student", 3.9);
    ptr -> print();
    delete ptr;

return 0;}
```



```cpp
class Student: public Person
{
    private:
        double gpa;
    public:
        Student (string name, double gpa);
        ~Student()
        {
            cout<<"Student's Destructor for "<<name<<endl;
        }
        void print ()
        {
            cout << "GPA: " << gpa << endl;
            cout << "Name: " << name << endl;
        }
};
Student :: Student (string nm, double gp)
: Person (nm), gpa (gp)
{
}
```



D:\Work Umair\4_CUST (1-9-22)\1_Teaching\5_CS1143-OOP\Practice Programs\

```
Name: Person
Person's Destructor for Person

GPA: 3.9
Name: Student
Student's Destructor for Student
Person's Destructor for Student
```

The destructor for Student class has been called now

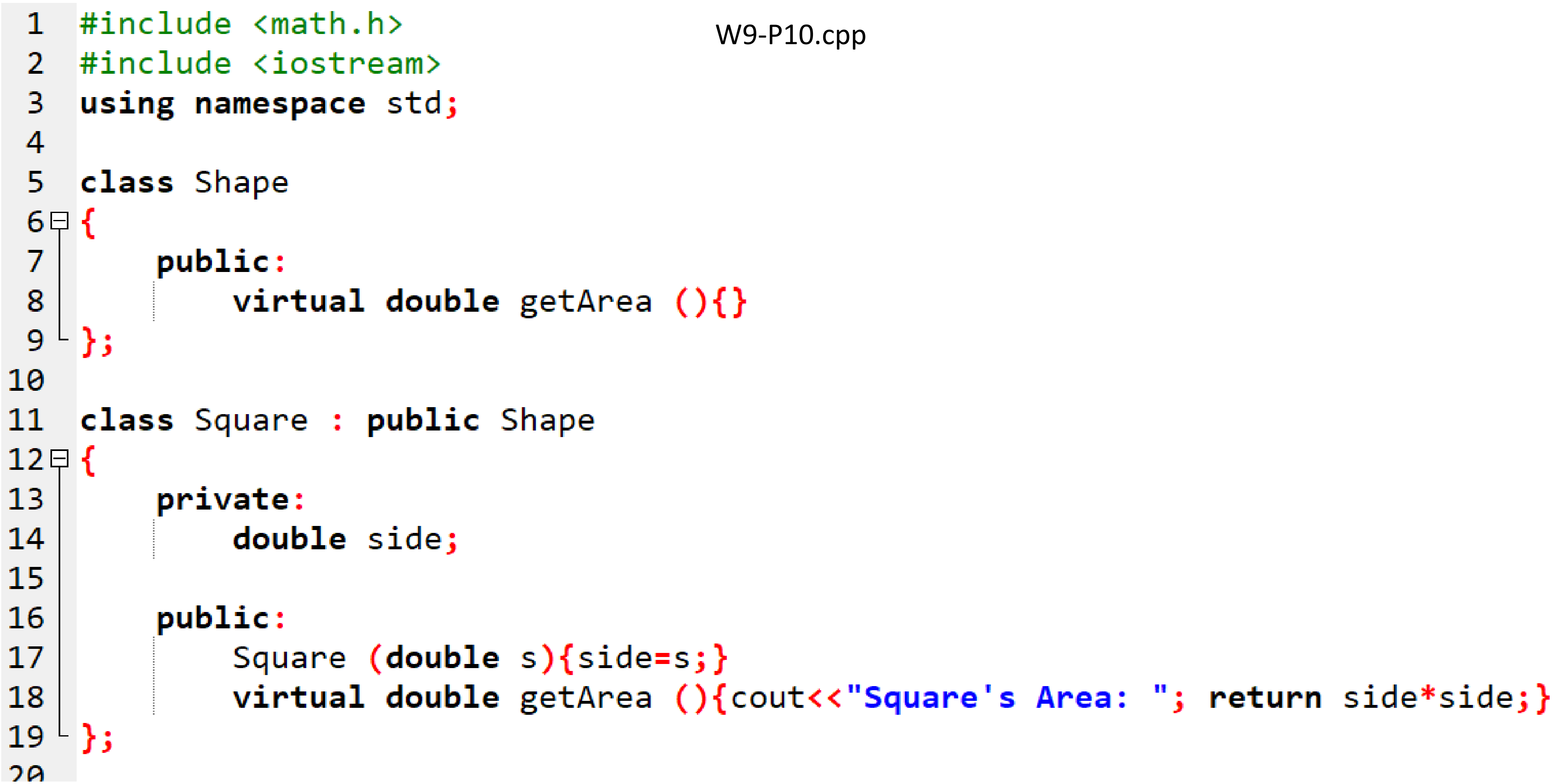
Example Program

The following program makes a class Shape and then inherits 3 shapes from it, Square, Circle and Rectangle.

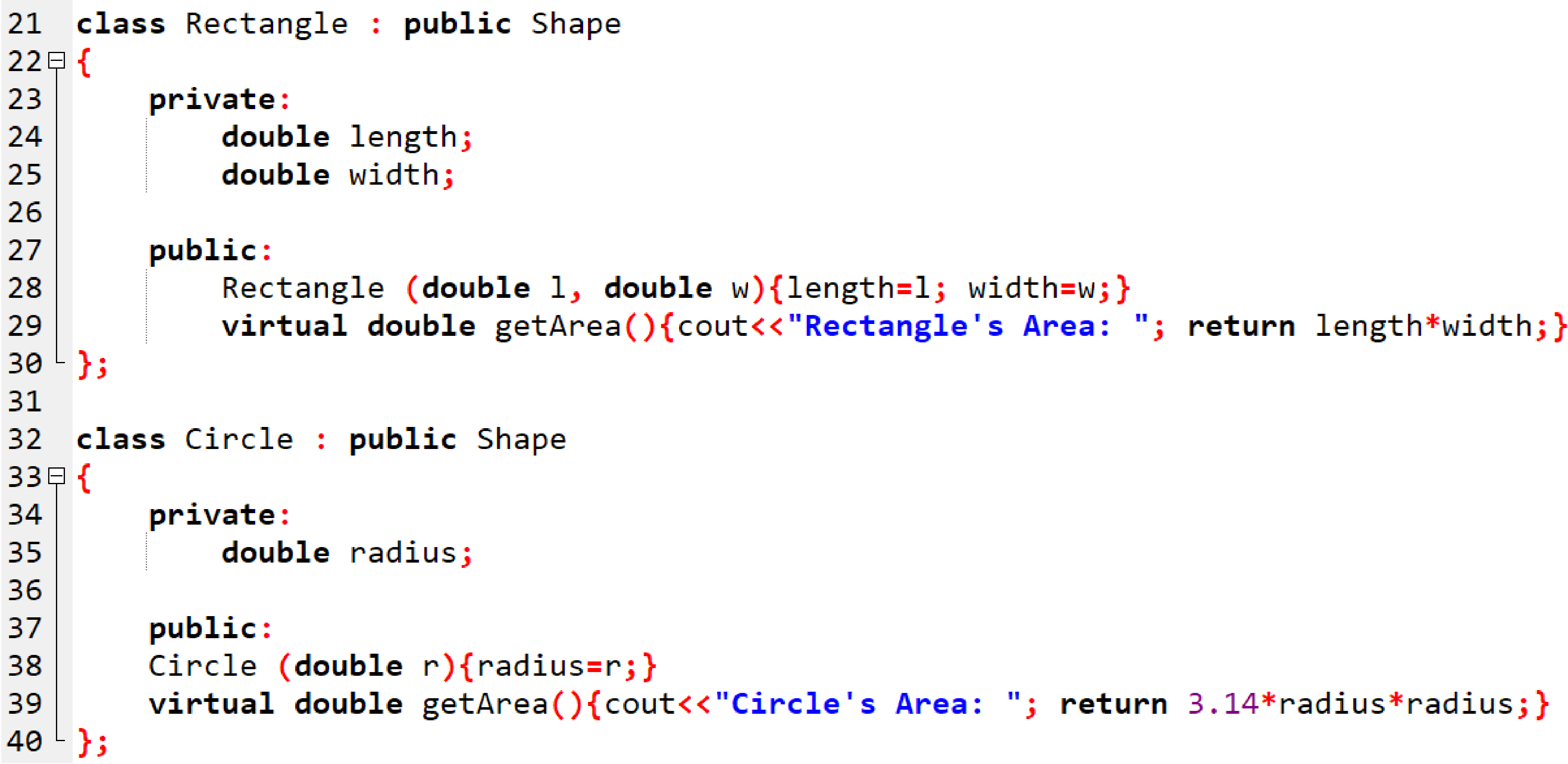
Each class overrides the function getArea() and has its own implementation of this function.

In the main function, we use polymorphism to call getArea().

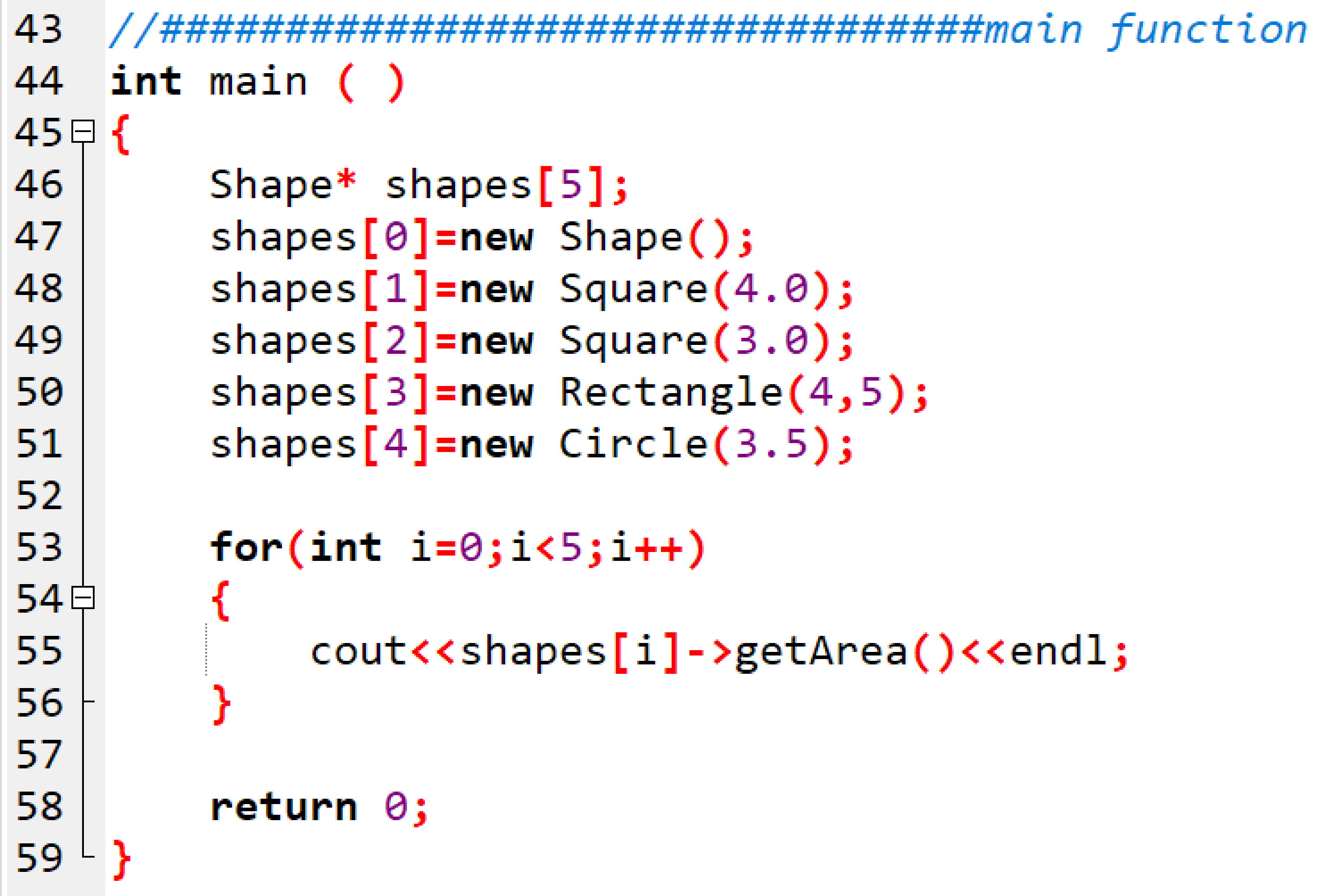
W9-P10.cpp



```cpp
#include <math.h>
#include <iostream>
using namespace std;

class Shape
{
    public:
        virtual double getArea (){}
};

class Square : public Shape
{
    private:
        double side;

    public:
        Square (double s){side=s;}
        virtual double getArea (){cout<<"Square's Area: "; return side*side;}
};
```



```
class Rectangle : public Shape
{
    private:
        double length;
        double width;

    public:
        Rectangle (double l, double w){length=l; width=w;}
        virtual double getArea(){cout<<"Rectangle's Area: "; return length*width;}
};

class Circle : public Shape
{
    private:
        double radius;

    public:
    Circle (double r){radius=r;}
    virtual double getArea(){cout<<"Circle's Area: "; return 3.14*radius*radius;}
};
```



```cpp
//#################################main function
int main ( )
{
    Shape* shapes[5];
    shapes[0]=new Shape();
    shapes[1]=new Square(4.0);
    shapes[2]=new Square(3.0);
    shapes[3]=new Rectangle(4,5);
    shapes[4]=new Circle(3.5);

    for(int i=0;i<5;i++)
    {
        cout<<shapes[i]->getArea()<<endl;
    }

    return 0;
}
```

This is all for Week 9