Polymorphism

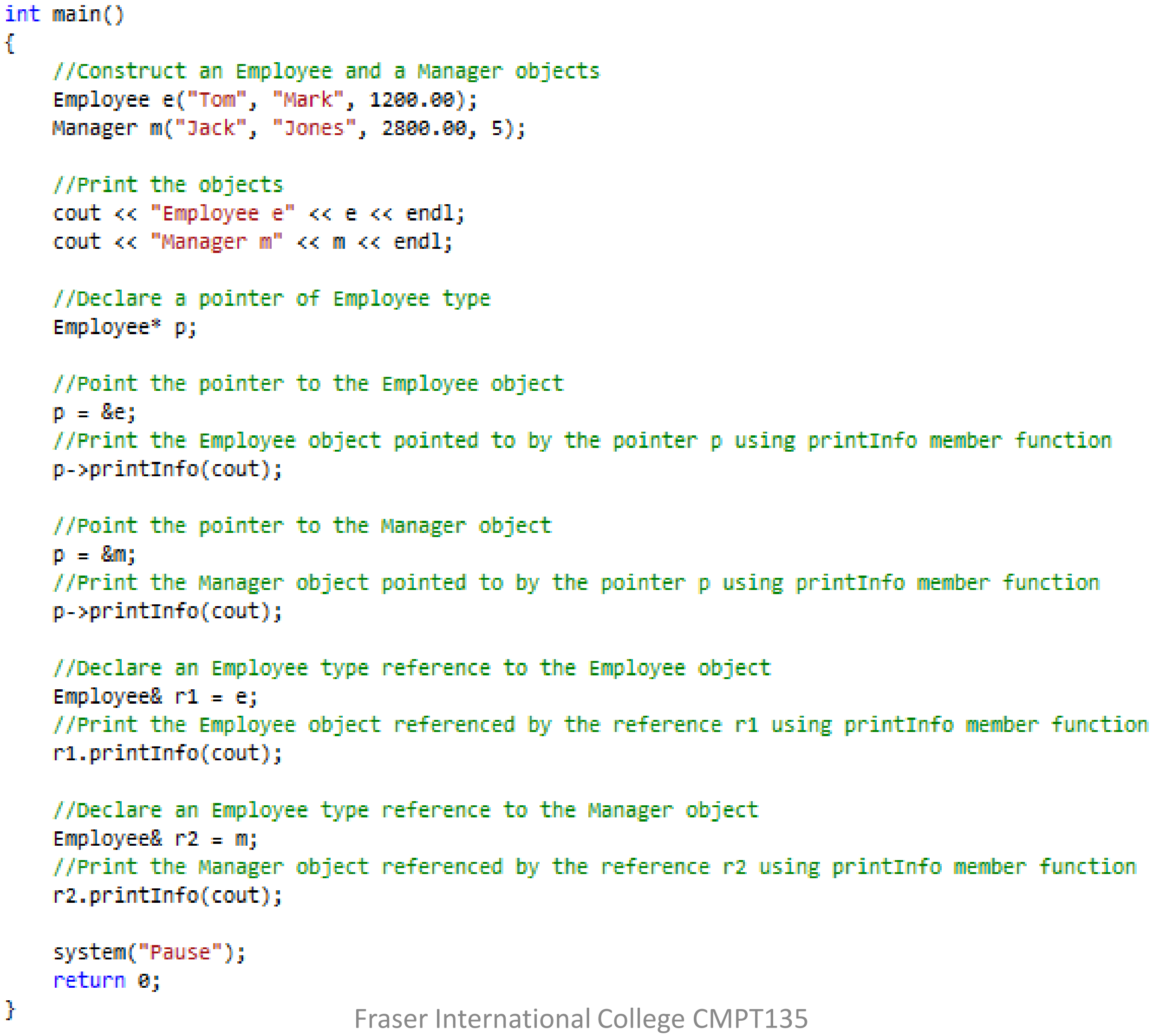
In this Week

- Polymorphism: Late Binding
- Virtual member functions
- Dynamic Cast
- Pure Virtual member functions
- Abstract classes and Interfaces

Motivation

- We consider once again the Employee base class and the Manager derived class developed during the topic of inheritance
- Our aim is to declare Employee type pointer variable and point it to Employee object or Manager object and call some member functions and see how we can make the pointer identify the object pointed by the pointer and call the correct member functions of the object
- Similarly we would like to declare Employee type reference variable and make the reference variable call the actual objects correct member functions
- See the following test program

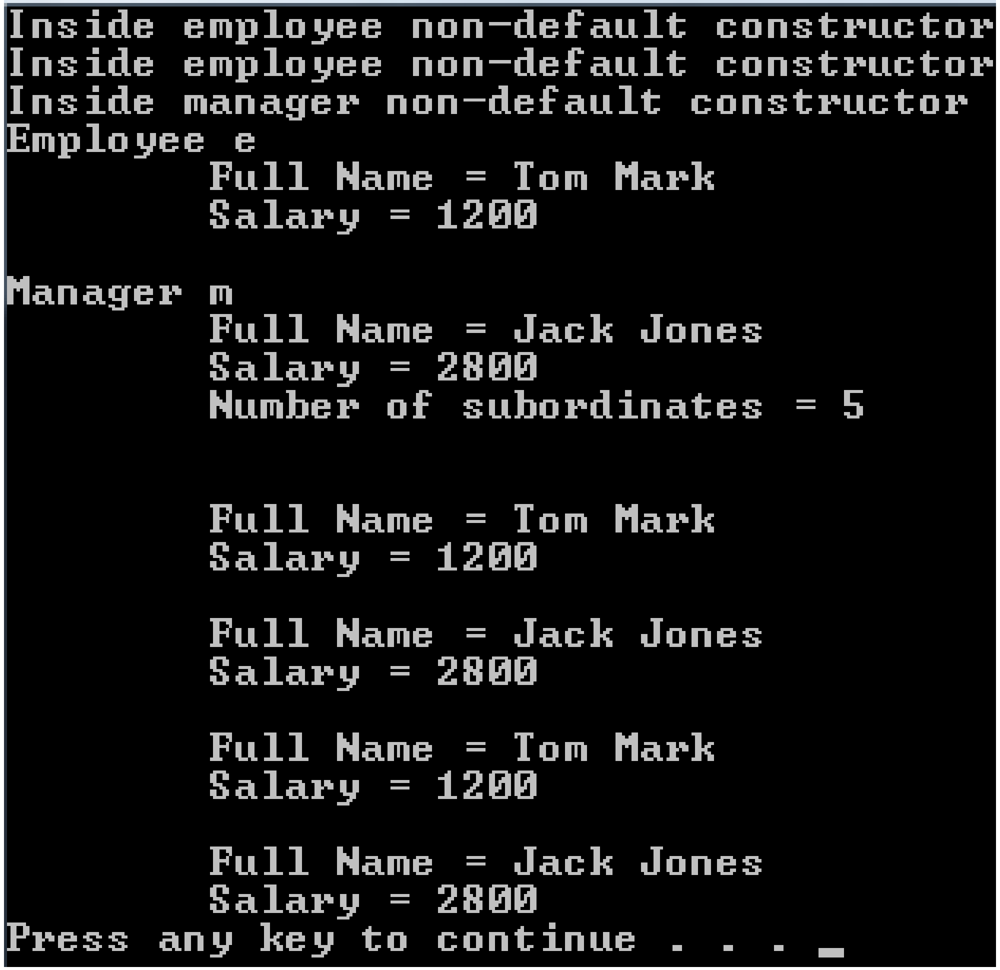
Motivation



```
int main()
{
    //Construct an Employee and a Manager objects
    Employee e("Tom", "Mark", 1200.00);
    Manager m("Jack", "Jones", 2800.00, 5);

    //Print the objects
    cout << "Employee e" << e << endl;
    cout << "Manager m" << m << endl;

    //Declare a pointer of Employee type
    Employee* p;

    //Point the pointer to the Employee object
    p = &e;
    //Print the Employee object pointed to by the pointer p using printInfo member function
    p->printInfo(cout);

    //Point the pointer to the Manager object
    p = &m;
    //Print the Manager object pointed to by the pointer p using printInfo member function
    p->printInfo(cout);

    //Declare an Employee type reference to the Employee object
    Employee& r1 = e;
    //Print the Employee object referenced by the reference r1 using printInfo member function
    r1.printInfo(cout);

    //Declare an Employee type reference to the Manager object
    Employee& r2 = m;
    //Print the Manager object referenced by the reference r2 using printInfo member function
    r2.printInfo(cout);

    system("Pause");
    return 0;
}
```

Motivation

- The output of the program will be as follows



```
Inside employee non-default constructor
Inside employee non-default constructor
Inside manager non-default constructor
Employee e
        Full Name = Tom Mark
        Salary = 1200

Manager m
        Full Name = Jack Jones
        Salary = 2800
        Number of subordinates = 5


        Full Name = Tom Mark
        Salary = 1200

        Full Name = Jack Jones
        Salary = 2800

        Full Name = Tom Mark
        Salary = 1200

        Full Name = Jack Jones
        Salary = 2800
Press any key to continue . . .
```

Motivation

- We notice that although the actual objects are not casted when we use pointers and references but still the pointer or reference variable are blindly calling the **printInfo** member function of the base class
- Now we ask ourselves how can we force the pointer or reference variable identify the underlying object and call its member functions
- **Answer:** We use **virtual member functions**

Virtual Member Functions

- In order to enforce runtime type checking (late binding) on pointers and references so that they check the underlying object type before executing a member function, we need to designate the member function virtual
- The **printInfo** member function therefore needs to designated as virtual
- The designation of virtual is needed to be done only on the class declaration but not on the class definition
- Moreover the designation of virtual is needed to be done only in the base class but it is not necessary to be designated as virtual in the derived class
- However C++ programmers typically designate member functions as virtual not only on the base class (which is required) but also on all the inheritance lineage for code clarity purposes
- The **printInfo** member function designated as virtual both on the base Employee class as well as the derived Manager class is shown below
- In Employee class

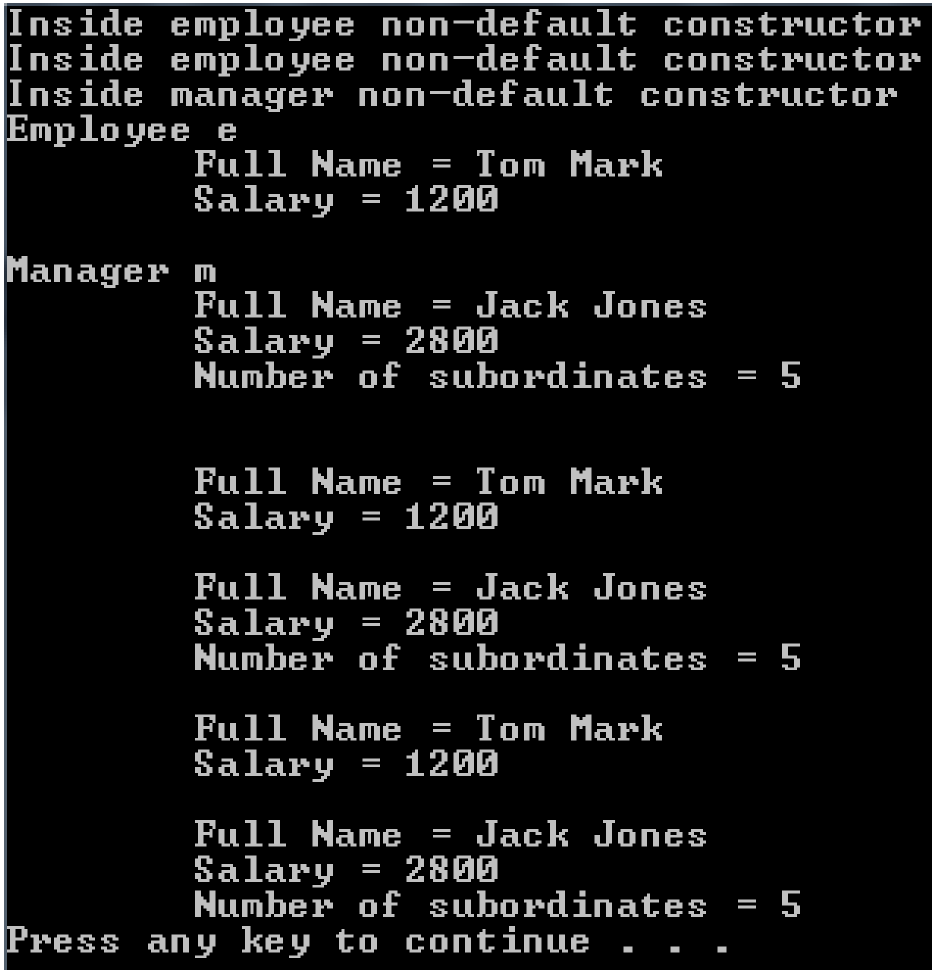
```
virtual void printInfo(ostream&) const;
```

- In Manager class

```
virtual void printInfo(ostream&) const;
```

Virtual Member Functions

- Now we execute the same test main program and its output will be



```
Inside employee non-default constructor
Inside employee non-default constructor
Inside manager non-default constructor
Employee e
        Full Name = Tom Mark
        Salary = 1200

Manager m
        Full Name = Jack Jones
        Salary = 2800
        Number of subordinates = 5


        Full Name = Tom Mark
        Salary = 1200

        Full Name = Jack Jones
        Salary = 2800
        Number of subordinates = 5

        Full Name = Tom Mark
        Salary = 1200

        Full Name = Jack Jones
        Salary = 2800
        Number of subordinates = 5
Press any key to continue . . .
```

Virtual Member Functions

- Thus during the design of classes, it is important to identify member functions that need to be overridden in derived classes and designate them as virtual member functions
- In our case, obviously the following modifications will be needed both in the Employee base class and the derived Manager class
- In Employee class

```
virtual ~Employee();
virtual void readInfo(istream&);
```

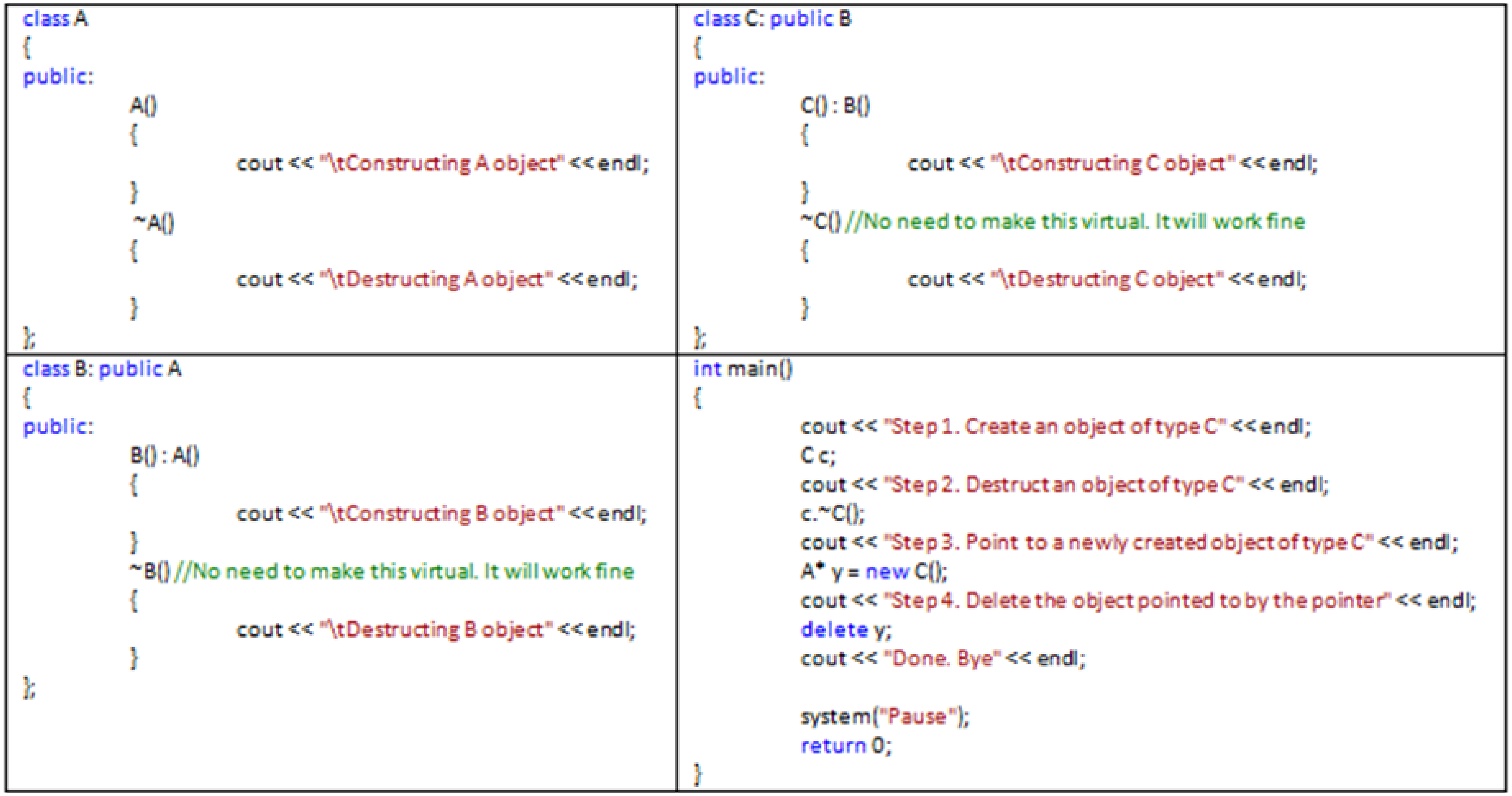
- In Manager class

```
virtual ~Manager();
virtual void readInfo(istream&);
```

- Observe that the **destructor must be designated virtual** otherwise we will end up destructing only base class part whenever we destruct objects using base class type pointers or references

Virtual Member Functions

- Analyze the following program and determine its output. What would be the effect of the destructor was designated virtual?



```
class A
{
public:
        A()
        {
                cout << "\tConstructing A object" << endl;
        }
        ~A()
        {
                cout << "\tDestructing A object" << endl;
        }
};
```

```
class B: public A
{
public:
        B() : A()
        {
                cout << "\tConstructing B object" << endl;
        }
        ~B() //No need to make this virtual. It will work fine
        {
                cout << "\tDestructing B object" << endl;
        }
};
```

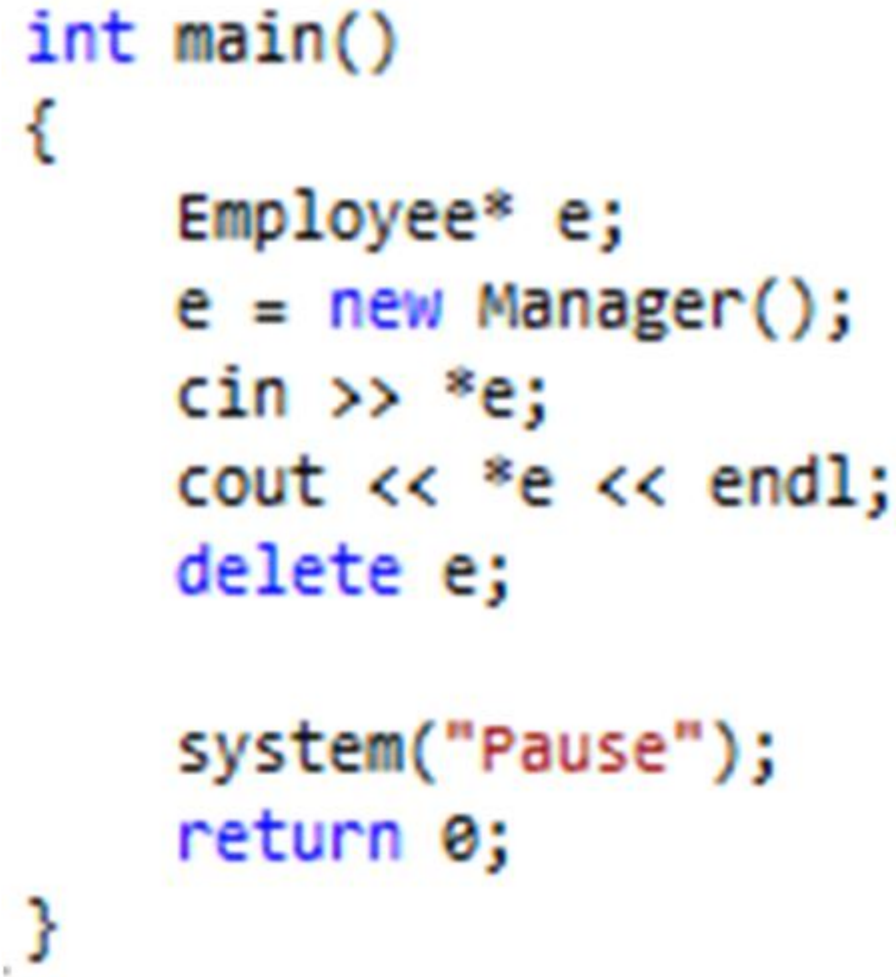
```
class C: public B
{
public:
        C() : B()
        {
                cout << "\tConstructing C object" << endl;
        }
        ~C() //No need to make this virtual. It will work fine
        {
                cout << "\tDestructing C object" << endl;
        }
};
```

```
int main()
{
        cout << "Step 1. Create an object of type C" << endl;
        C c;
        cout << "Step 2. Destruct an object of type C" << endl;
        c.~C();
        cout << "Step 3. Point to a newly created object of type C" << endl;
        A* y = new C();
        cout << "Step 4. Delete the object pointed to by the pointer" << endl;
        delete y;
        cout << "Done. Bye" << endl;

        system("Pause");
        return 0;
}
```

Virtual Member Functions

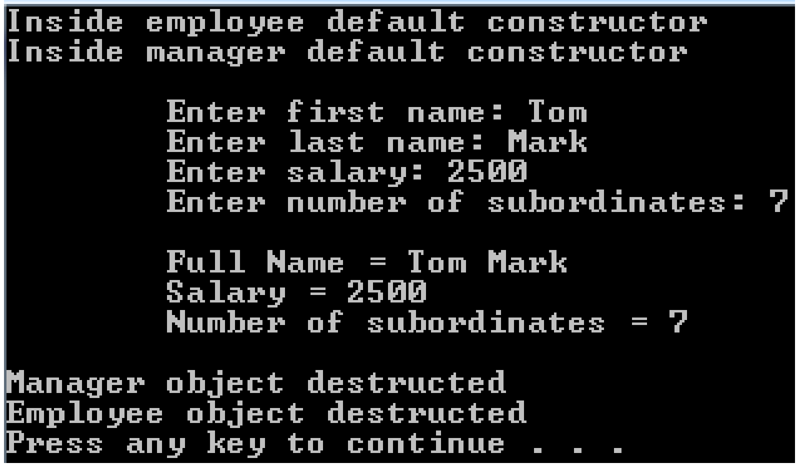
- With the design we now have, we can actually remove the code for the istream and ostream friend operator functions from the Manager class without affecting anything in the workings of our classes
- How?
- Because the istream and ostream friend operator functions in the Employee class will be sufficient for after all when we print Manager objects, these objects will pass by reference to the istream and ostream friend operator functions in the Employee class and because we are passing by reference then the parameter objects will call their correct readInfo or printInfo member functions
- The following program demonstrates the effect of virtual functions and their effects on objects

Virtual Member Functions



```
int main()
{
    Employee* e;
    e = new Manager();
    cin >> *e;
    cout << *e << endl;
    delete e;

    system("Pause");
    return 0;
}
```

Sample run output

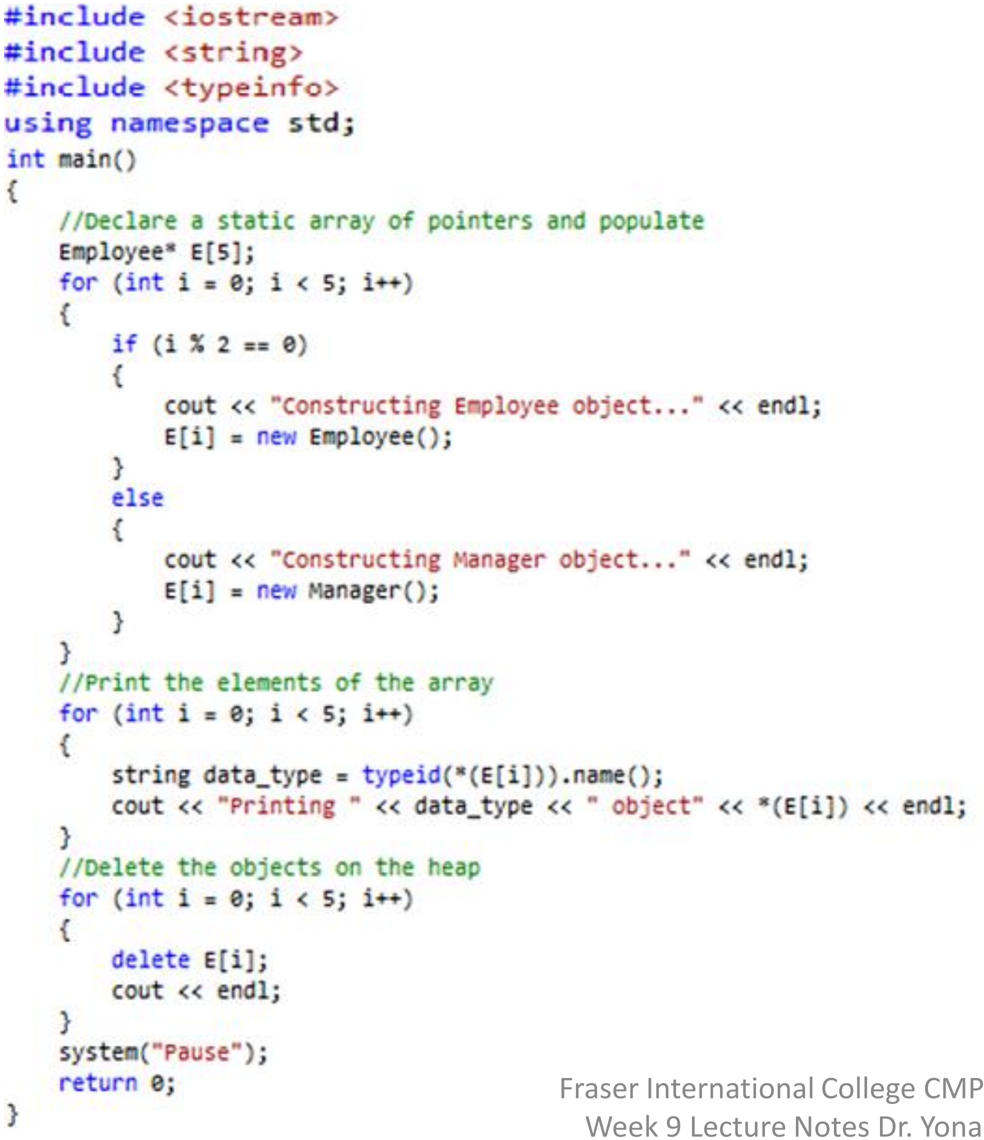


```
Inside employee default constructor
Inside manager default constructor

        Enter first name: Tom
        Enter last name: Mark
        Enter salary: 2500
        Enter number of subordinates: 7

        Full Name = Tom Mark
        Salary = 2500
        Number of subordinates = 7

Manager object destructed
Employee object destructed
Press any key to continue . . .
```

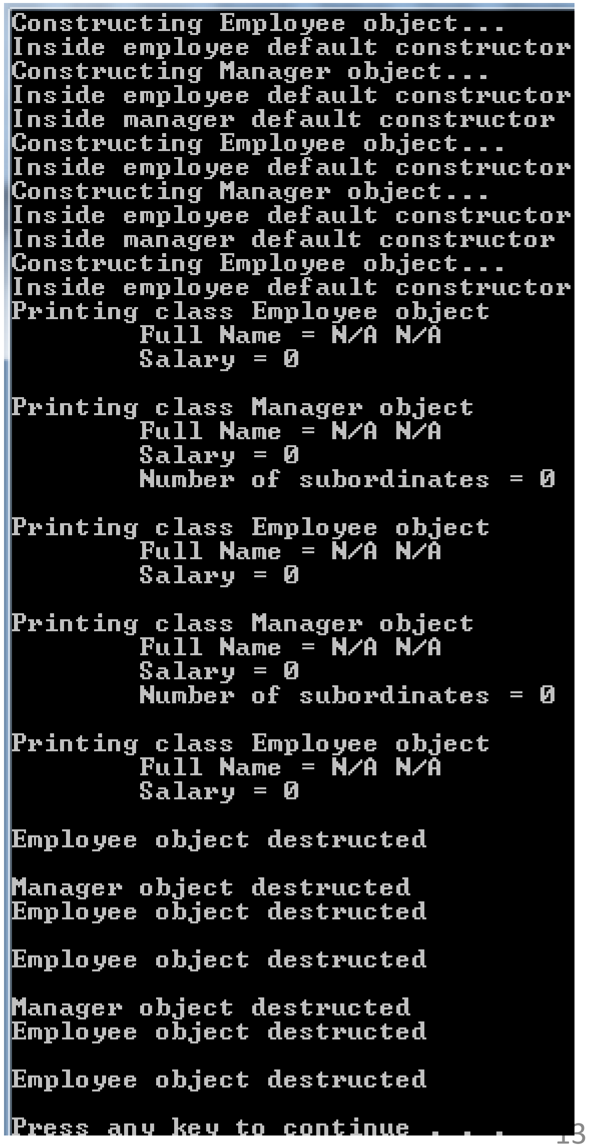
Polymorphism

- The process of enforcing **late binding** (runtime type checking) on pointers and references is known as polymorphism
- With the help of polymorphism, we can now create an array of base class type pointers and point the elements of the array to base class or derived class objects without any casting taking place and finally traverse the array and invoke some functions on the elements of the array and so long as we call virtual member functions then the elements of the array will intelligently call the underlying objects' correct member functions as shown below
- We also show how to use the **typeid built-in function** in the **typeinfo library** in order to have informative messages in our output

Polymorphism



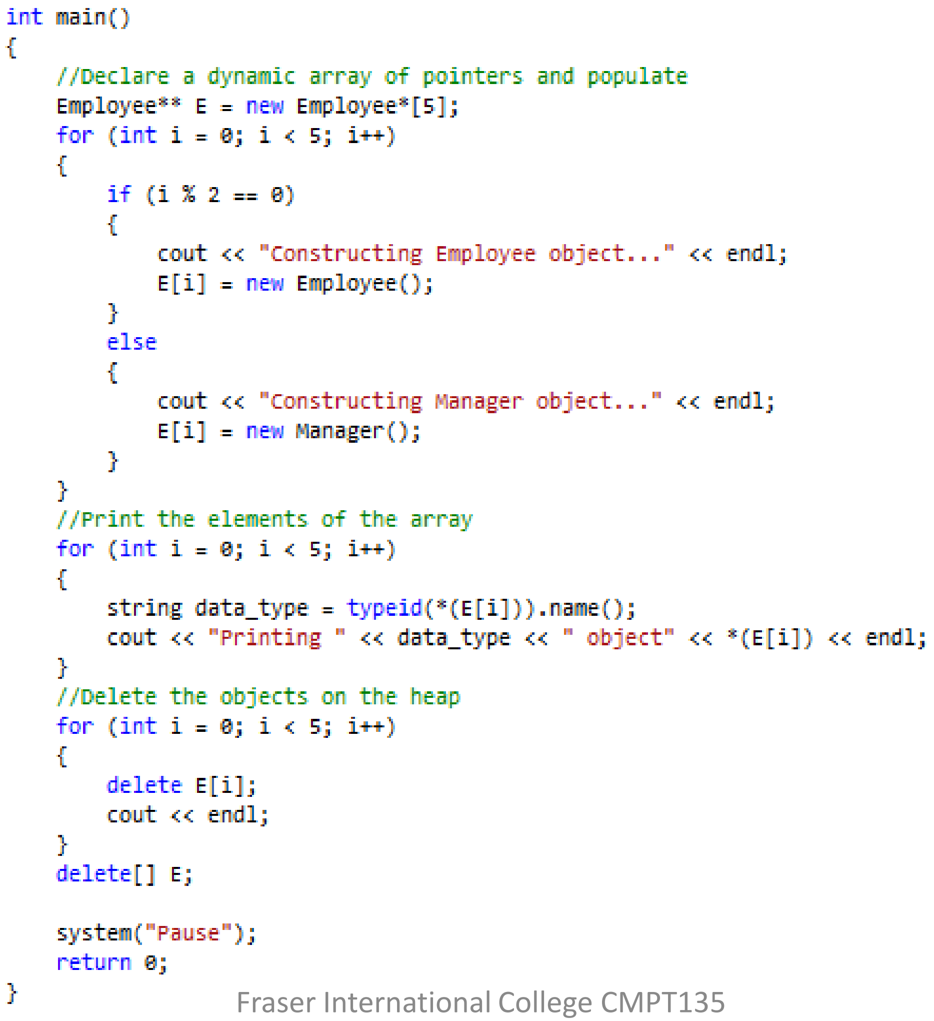
```cpp
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;
int main()
{
    //Declare a static array of pointers and populate
    Employee* E[5];
    for (int i = 0; i < 5; i++)
    {
        if (i % 2 == 0)
        {
            cout << "Constructing Employee object..." << endl;
            E[i] = new Employee();
        }
        else
        {
            cout << "Constructing Manager object..." << endl;
            E[i] = new Manager();
        }
    }
    //Print the elements of the array
    for (int i = 0; i < 5; i++)
    {
        string data_type = typeid(*(E[i])).name();
        cout << "Printing " << data_type << " object" << *(E[i]) << endl;
    }
    //Delete the objects on the heap
    for (int i = 0; i < 5; i++)
    {
        delete E[i];
        cout << endl;
    }
    system("Pause");
    return 0;
}
```



```
Constructing Employee object...
Inside employee default constructor
Constructing Manager object...
Inside employee default constructor
Inside manager default constructor
Constructing Employee object...
Inside employee default constructor
Constructing Manager object...
Inside employee default constructor
Inside manager default constructor
Constructing Employee object...
Inside employee default constructor
Printing class Employee object
        Full Name = N/A N/A
        Salary = 0

Printing class Manager object
        Full Name = N/A N/A
        Salary = 0
        Number of subordinates = 0

Printing class Employee object
        Full Name = N/A N/A
        Salary = 0

Printing class Manager object
        Full Name = N/A N/A
        Salary = 0
        Number of subordinates = 0

Printing class Employee object
        Full Name = N/A N/A
        Salary = 0

Employee object destructed

Manager object destructed
Employee object destructed

Employee object destructed

Manager object destructed
Employee object destructed

Employee object destructed

Press any key to continue . . .
```

Polymorphism

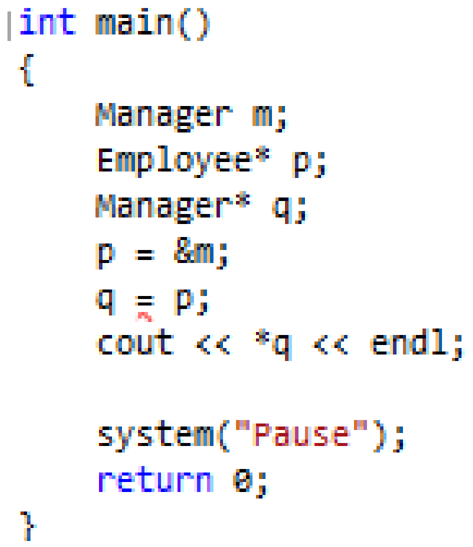
- Similarly we may use dynamic array as follows and it will perform the same way



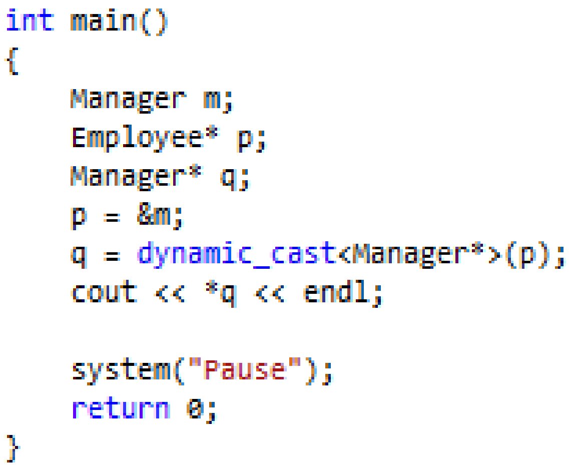
```
int main()
{
    //Declare a dynamic array of pointers and populate
    Employee** E = new Employee*[5];
    for (int i = 0; i < 5; i++)
    {
        if (i % 2 == 0)
        {
            cout << "Constructing Employee object..." << endl;
            E[i] = new Employee();
        }
        else
        {
            cout << "Constructing Manager object..." << endl;
            E[i] = new Manager();
        }
    }
    //Print the elements of the array
    for (int i = 0; i < 5; i++)
    {
        string data_type = typeid(*(E[i])).name();
        cout << "Printing " << data_type << " object" << *(E[i]) << endl;
    }
    //Delete the objects on the heap
    for (int i = 0; i < 5; i++)
    {
        delete E[i];
        cout << endl;
    }
    delete[] E;

    system("Pause");
    return 0;
}
```

Dynamic Cast

- Consider the following program (top part)
- We would think the assignment statement **q = p;** is valid because **p** is pointing to a Manager object
- But we see that the assignment statement has syntax error. But why? Because the compiler can not ascertain the pointer **p** will be pointing to a Manager object when the program executes
- Remember if **p** were pointing to an Employee object then we should expect an error
- One way to inform the compiler that the type check has to be performed during program execution is to use **dynamic_cast**
- That way the compiler will leave the type check to be performed during run time
- The same program with a dynamic cast is shown (bottom part)
- We should note that dynamic cast causes run time error if the pointer to be casted is not pointing to a Manager object



```
int main()
{
        Manager m;
        Employee* p;
        Manager* q;
        p = &m;
        q = p;
        cout << *q << endl;

        system("Pause");
        return 0;
}
```



```
int main()
{
    Manager m;
    Employee* p;
    Manager* q;
    p = &m;
    q = dynamic_cast<Manager*>(p);
    cout << *q << endl;

    system("Pause");
    return 0;
}
```

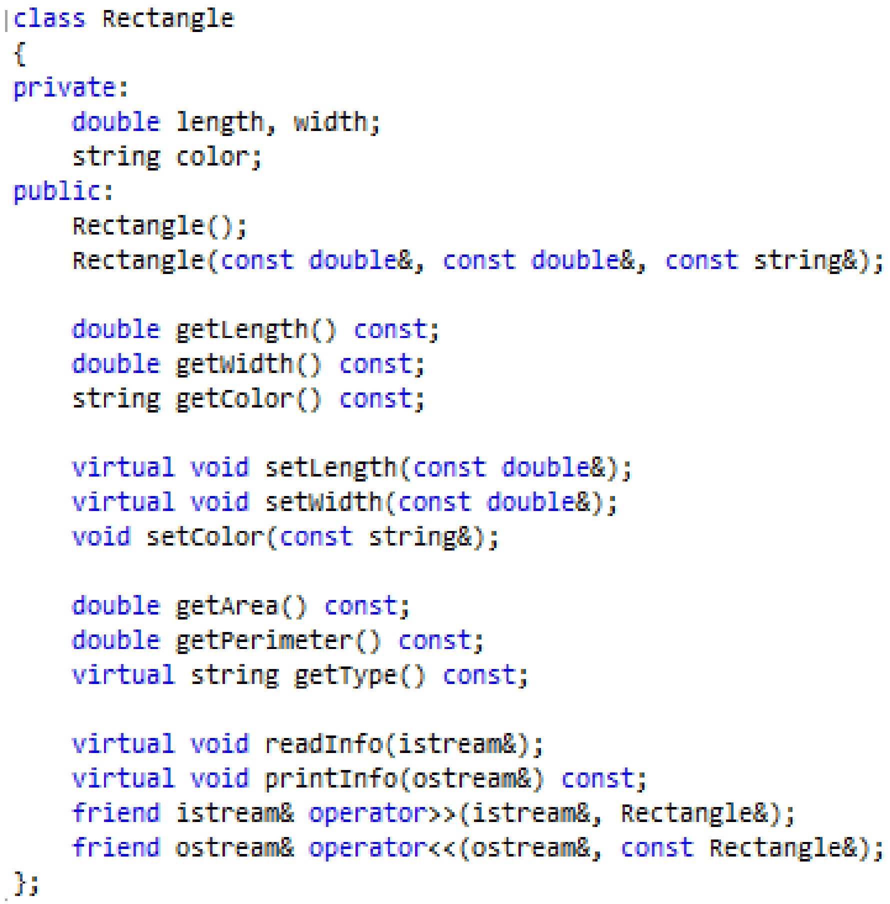
Dynamic Cast

- We could also use static_cast instead of dynamic_cast
- However while dynamic_cast will perform validity check before casting and thus we may catch the error during runtime, static_cast will not perform validity check and will cause runtime error
- As a last remark, for a dynamic_cast to be a valid operation, there has to be at least one virtual member function in the base class which is to say the class inheritance must have polymorphic behavior
- We conclude that whenever base class pointers are used in our applications, then unless we use virtual member functions they will always call their class type member functions irrespective of the actual objects they are pointing to and this may be semantically incorrect
- In particular, it is a good programming habit to always designate a destructor of a class virtual to guarantee correct destruction

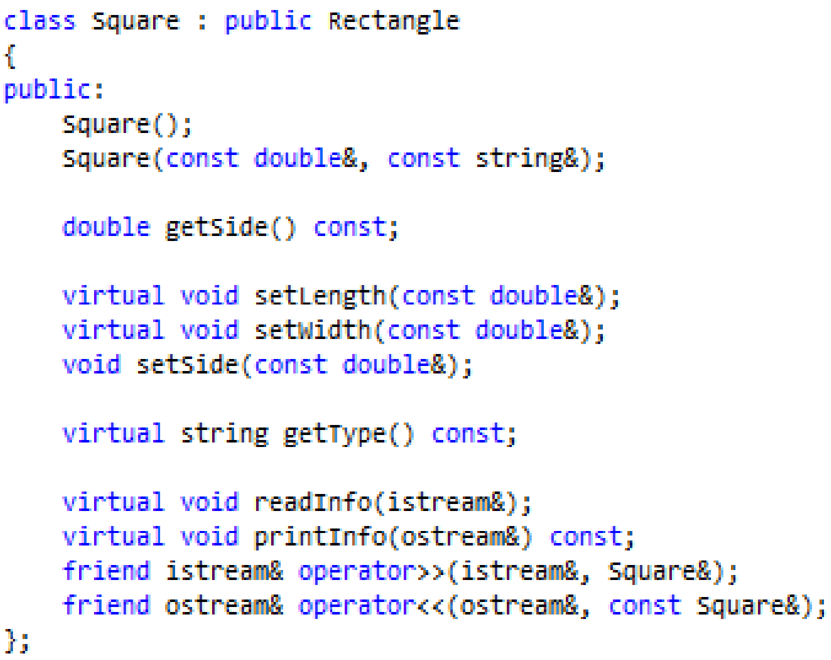
Pure Virtual Functions
Abstract Classes and Interfaces

- Now consider the following problem statement
- **Problem Statement:** Design classes and their inheritance relationships to represent **`Rectangle`**, **`Square`**, **`Triangle`**, and **`Circle`** objects. Assume each of such objects will have a color attribute in addition to its geometrical attributes and provide all the required constructors, destructors, getters, setters, and any other member functions. In particular, provide **`getColor`**, **`getType`** (to return the data type of an object as a string), **`getArea`**, **`getCircumference`**, **`readInfo`**, and **`printInfo`** member functions. Last but not least, design your classes such that at the end a container of your base class data type may store several objects of any type of geometrical objects represented by your classes
- We may therefore design the classes as follows

Pure Virtual Functions
Abstract Classes and Interfaces

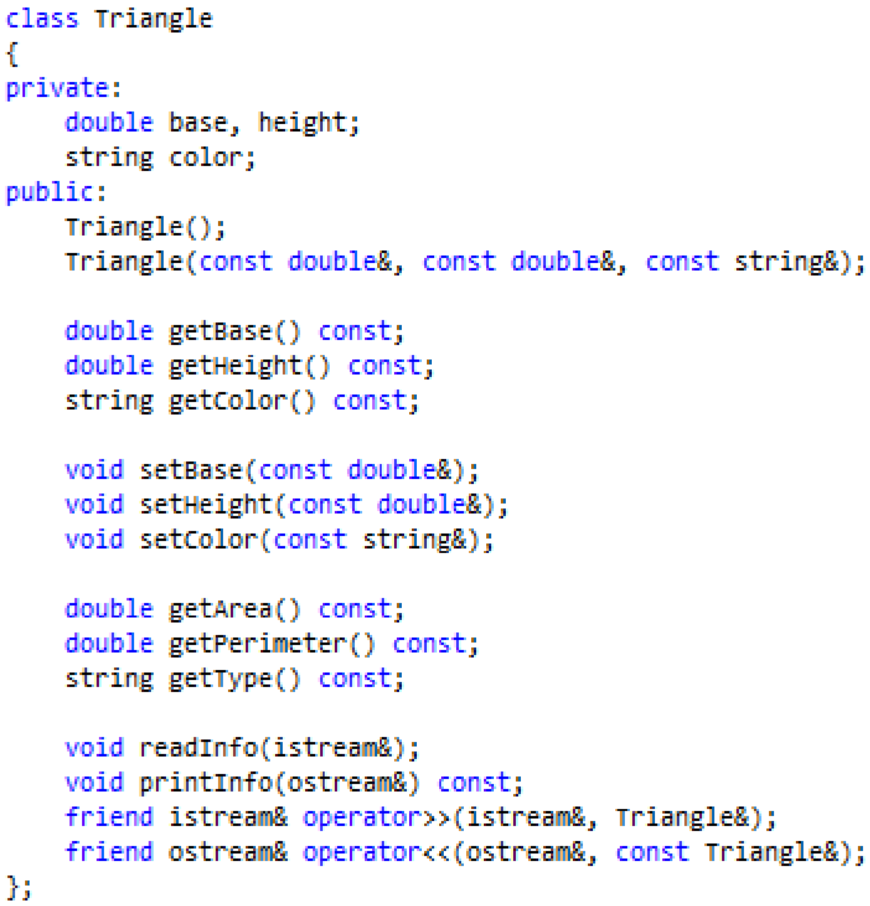


```
class Rectangle
{
private:
    double length, width;
    string color;
public:
    Rectangle();
    Rectangle(const double&, const double&, const string&);

    double getLength() const;
    double getWidth() const;
    string getColor() const;

    virtual void setLength(const double&);
    virtual void setWidth(const double&);
    void setColor(const string&);

    double getArea() const;
    double getPerimeter() const;
    virtual string getType() const;

    virtual void readInfo(istream&);
    virtual void printInfo(ostream&) const;
    friend istream& operator>>(istream&, Rectangle&);
    friend ostream& operator<<(ostream&, const Rectangle&);
};
```

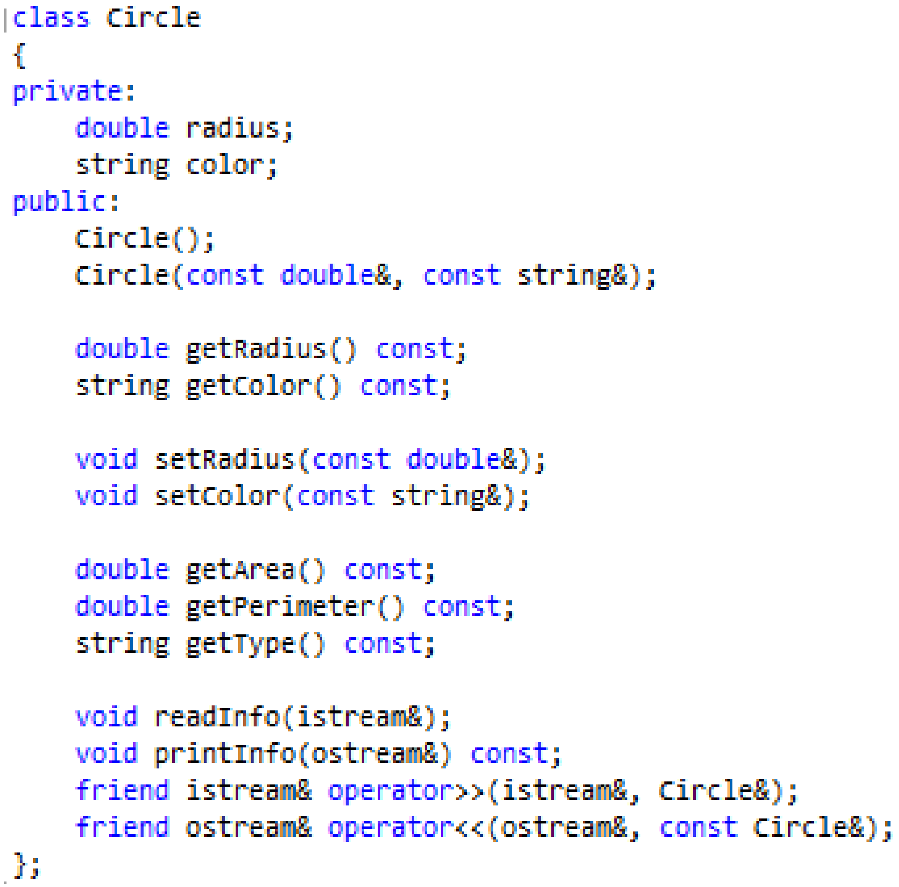


```
class Square : public Rectangle
{
public:
    Square();
    Square(const double&, const string&);

    double getSide() const;

    virtual void setLength(const double&);
    virtual void setWidth(const double&);
    void setSide(const double&);

    virtual string getType() const;

    virtual void readInfo(istream&);
    virtual void printInfo(ostream&) const;
    friend istream& operator>>(istream&, Square&);
    friend ostream& operator<<(ostream&, const Square&);
};
```

Pure Virtual Functions
Abstract Classes and Interfaces



```cpp
class Triangle
{
private:
    double base, height;
    string color;
public:
    Triangle();
    Triangle(const double&, const double&, const string&);

    double getBase() const;
    double getHeight() const;
    string getColor() const;

    void setBase(const double&);
    void setHeight(const double&);
    void setColor(const string&);

    double getArea() const;
    double getPerimeter() const;
    string getType() const;

    void readInfo(istream&);
    void printInfo(ostream&) const;
    friend istream& operator>>(istream&, Triangle&);
    friend ostream& operator<<(ostream&, const Triangle&);
};
```

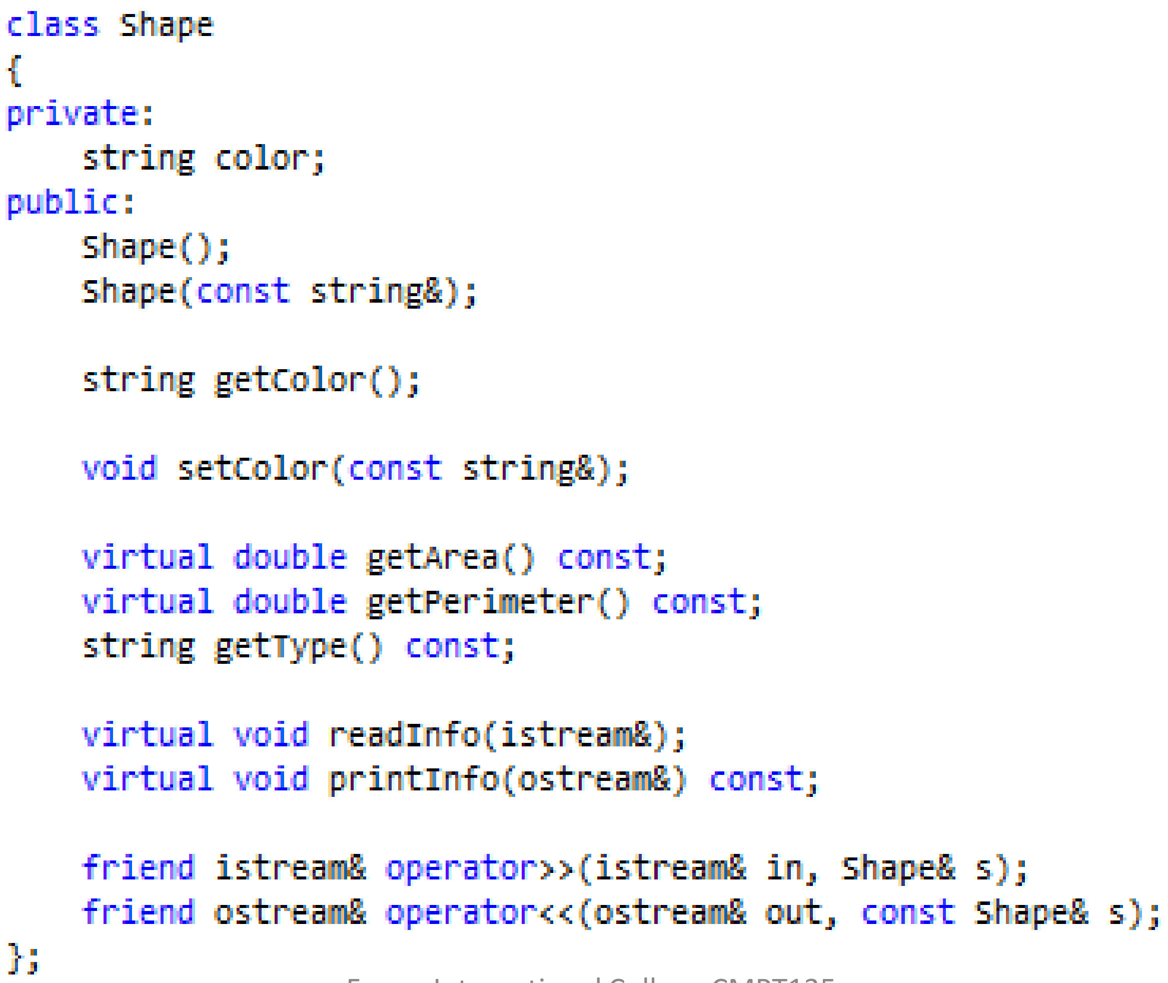


```cpp
class Circle
{
private:
    double radius;
    string color;
public:
    Circle();
    Circle(const double&, const string&);

    double getRadius() const;
    string getColor() const;

    void setRadius(const double&);
    void setColor(const string&);

    double getArea() const;
    double getPerimeter() const;
    string getType() const;

    void readInfo(istream&);
    void printInfo(ostream&) const;
    friend istream& operator>>(istream&, Circle&);
    friend ostream& operator<<(ostream&, const Circle&);
};
```

Pure Virtual Functions Abstract Classes and Interfaces

- With the design we now have, we will still need a base class for all the classes so that a base class type container will be able to store any object constructed from our classes
- Let us design a base class named **Shape**
- We therefore now move all the common attributes (member variables or member functions) of the different classes to the base class and designate them virtual so that each class will implement them in a correct way for the objects it represents
- The following class declaration shows the **Shape** class declaration as we would start to design it

Pure Virtual Functions
Abstract Classes and Interfaces



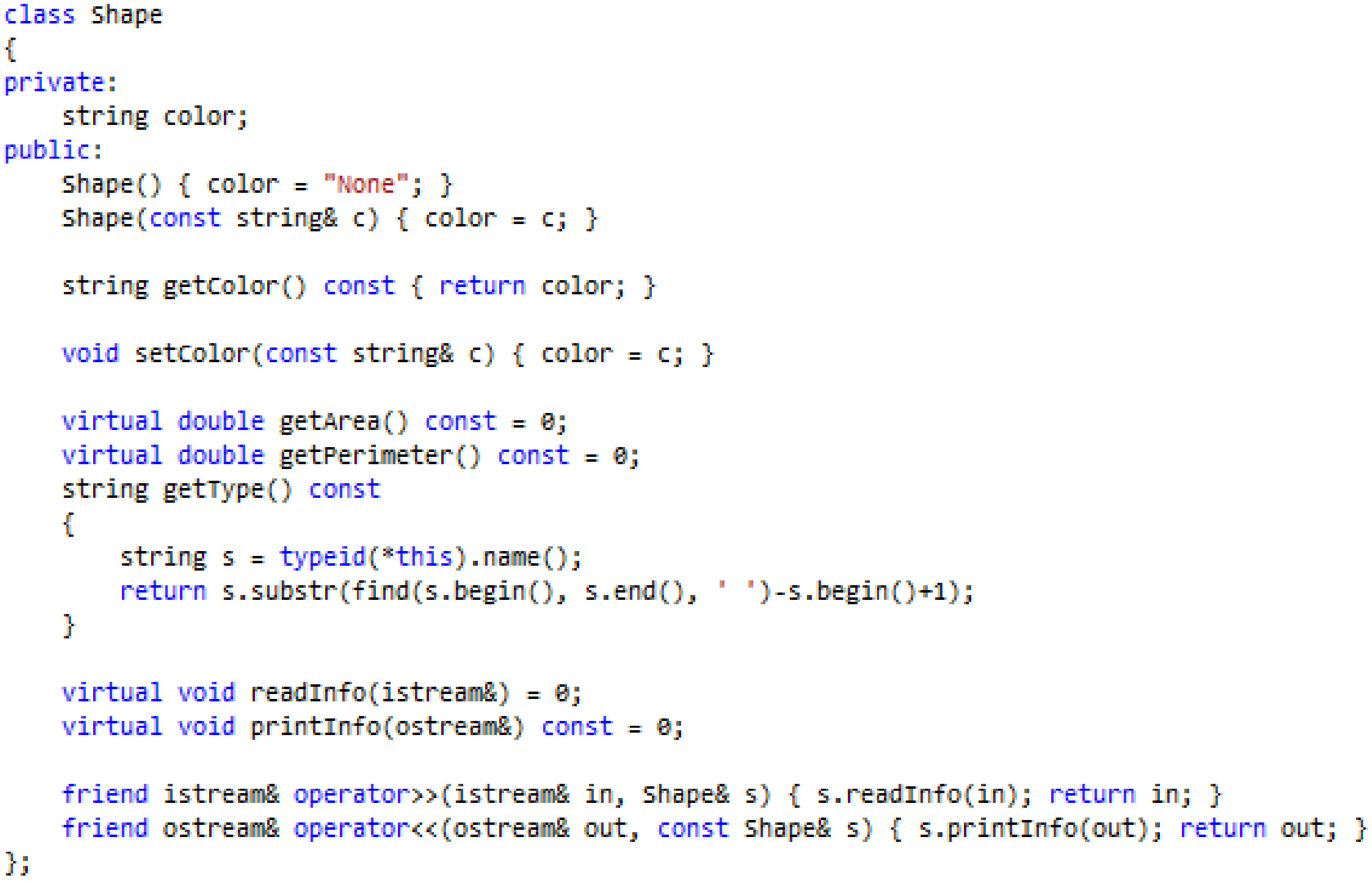
```
class Shape
{
private:
    string color;
public:
    Shape();
    Shape(const string&);

    string getColor();

    void setColor(const string&);

    virtual double getArea() const;
    virtual double getPerimeter() const;
    string getType() const;

    virtual void readInfo(istream&);
    virtual void printInfo(ostream&) const;

    friend istream& operator>>(istream& in, Shape& s);
    friend ostream& operator<<(ostream& out, const Shape& s);
};
```

Pure Virtual Functions Abstract Classes and Interfaces

- Next, we start implementing the **`Shape`** class member functions
- The constructors and getters will be straightforward. But what about the **`getArea`** and **`getPerimeter`** member functions?
- We observe that the **`Shape`** class does not actually represent concrete objects; rather it is a blue print for other concrete objects such as Rectangle, Square, Triangle and Square objects
- This implies the **`getArea`** and **`getPerimeter`** member functions should be declared but they should not be defined in the **`Shape`** class
- But C++ requires every declared member function to be implemented
- This is where pure virtual member functions come to play a role
- Whenever a base class does not need to implement any member function but rather leave the implementation to derived classes, then such functions should be designated as pure virtual in the base class
- A member function in a base class is designated as pure virtual by placing = 0 in its declaration
- It is not necessary for a derived class to implement a pure virtual member function. But if it doesn't then there will be consequences discussed later

Pure Virtual Functions
Abstract Classes and Interfaces

- The **Shape** class with its pure virtual member functions together with implementations of the remaining member functions is shown below

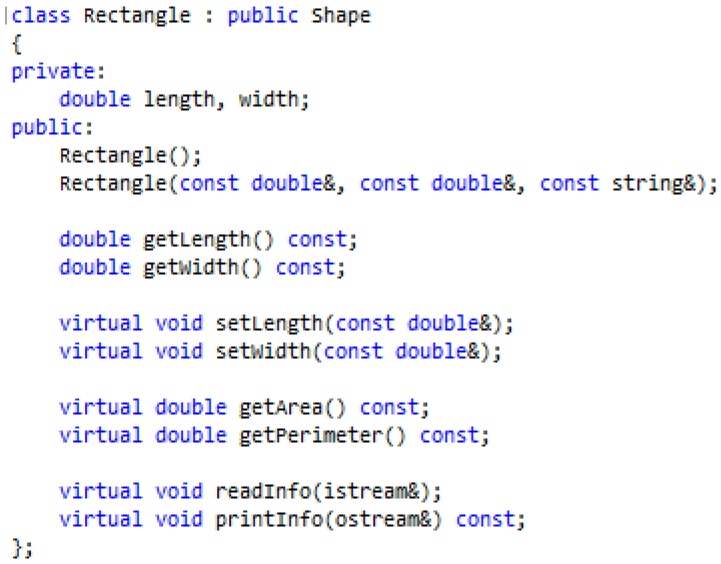


```
class Shape
{
private:
    string color;
public:
    Shape() { color = "None"; }
    Shape(const string& c) { color = c; }

    string getColor() const { return color; }

    void setColor(const string& c) { color = c; }

    virtual double getArea() const = 0;
    virtual double getPerimeter() const = 0;
    string getType() const
    {
        string s = typeid(*this).name();
        return s.substr(find(s.begin(), s.end(), ' ')-s.begin()+1);
    }

    virtual void readInfo(istream&) = 0;
    virtual void printInfo(ostream&) const = 0;

    friend istream& operator>>(istream& in, Shape& s) { s.readInfo(in); return in; }
    friend ostream& operator<<(ostream& out, const Shape& s) { s.printInfo(out); return out; }
};
```

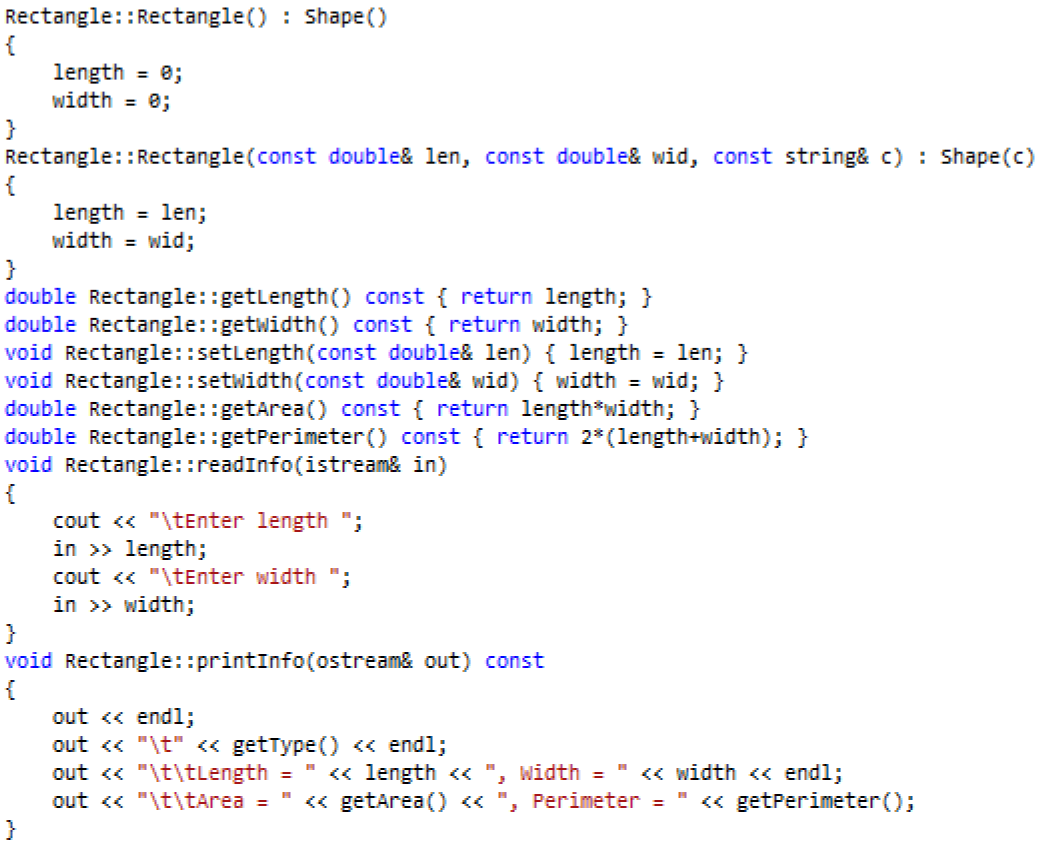
Pure Virtual Functions Abstract Classes and Interfaces

- A C++ class with at least one pure virtual member function is known as abstract class
- Thus the Shape class shown above is an abstract class
- We can not create (instantiate) an object of an abstract class type
- Neither can an abstract class type used as a parameter if the parameter pass is by value
- However pointers and references of abstract classes can be declared and used with concrete objects
- If a derived class does not implement any of the pure virtual member functions in its base class then the derived class is also automatically an abstract class and the constraints of an abstract class (such as not being able to instantiate an object of the class type) will apply to it
- A C++ class whose member functions are all pure virtual member functions is known as an interface
- The declarations of the Rectangle, Square, Triangle and Circle classes modified to be derived from the Shape class together with their implementations are shown below

Pure Virtual Functions
Abstract Classes and Interfaces

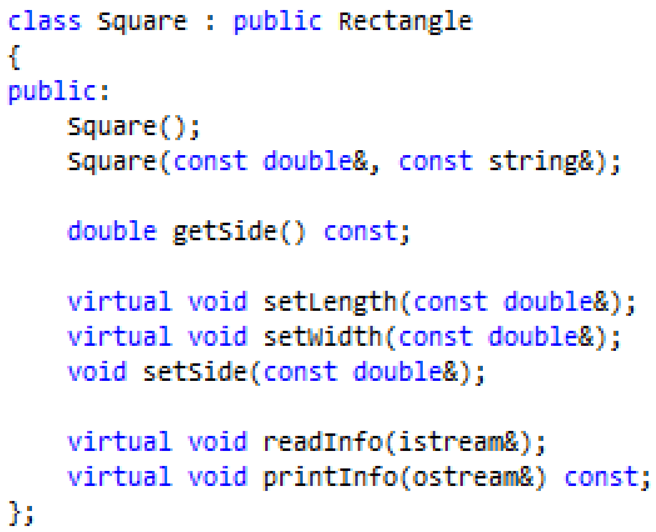


```
class Rectangle : public Shape
{
private:
    double length, width;
public:
    Rectangle();
    Rectangle(const double&, const double&, const string&);

    double getLength() const;
    double getWidth() const;

    virtual void setLength(const double&);
    virtual void setWidth(const double&);

    virtual double getArea() const;
    virtual double getPerimeter() const;

    virtual void readInfo(istream&);
    virtual void printInfo(ostream&) const;
};
```

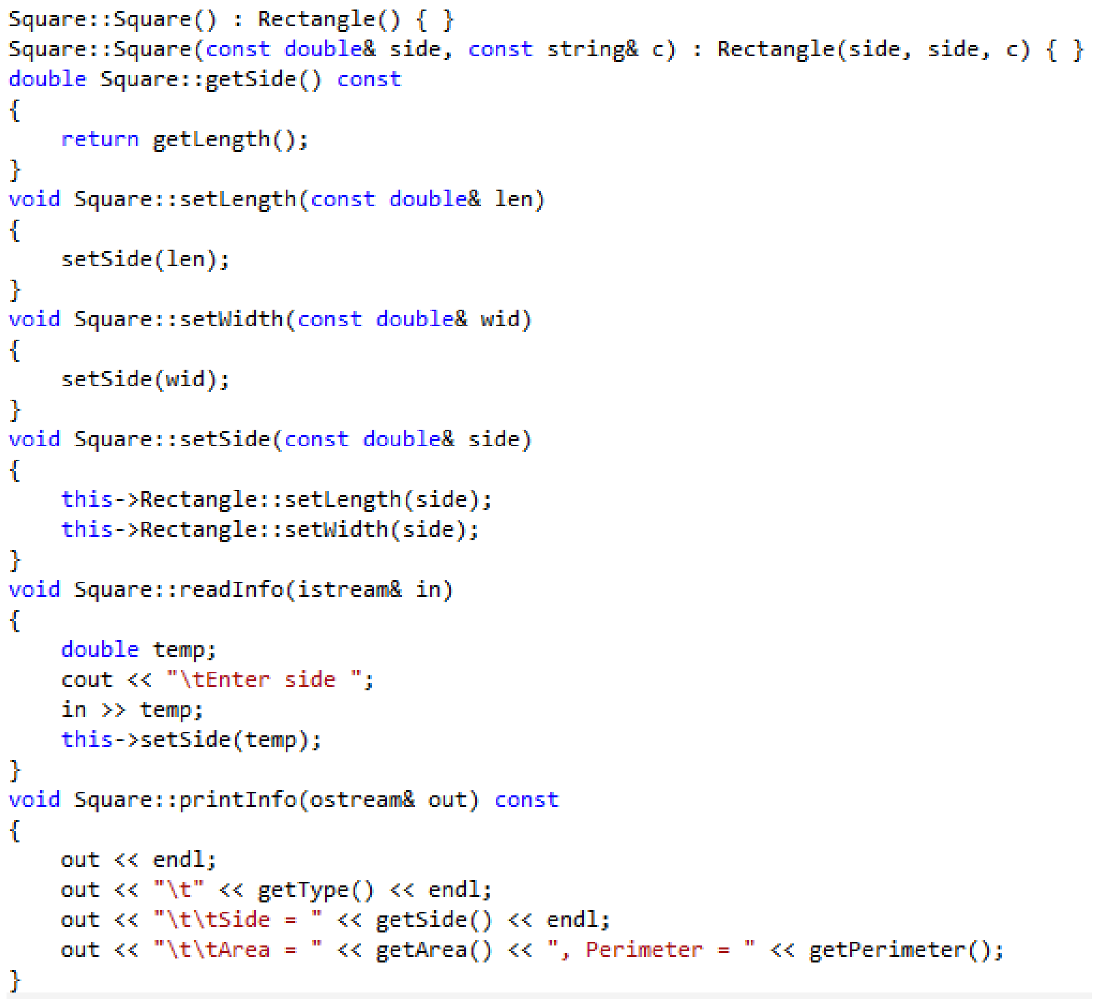


```
Rectangle::Rectangle() : Shape()
{
    length = 0;
    width = 0;
}
Rectangle::Rectangle(const double& len, const double& wid, const string& c) : Shape(c)
{
    length = len;
    width = wid;
}
double Rectangle::getLength() const { return length; }
double Rectangle::getWidth() const { return width; }
void Rectangle::setLength(const double& len) { length = len; }
void Rectangle::setWidth(const double& wid) { width = wid; }
double Rectangle::getArea() const { return length*width; }
double Rectangle::getPerimeter() const { return 2*(length+width); }
void Rectangle::readInfo(istream& in)
{
    cout << "\tEnter length ";
    in >> length;
    cout << "\tEnter width ";
    in >> width;
}
void Rectangle::printInfo(ostream& out) const
{
    out << endl;
    out << "\t" << getType() << endl;
    out << "\t\tLength = " << length << ", Width = " << width << endl;
    out << "\t\tArea = " << getArea() << ", Perimeter = " << getPerimeter();
}
```

Pure Virtual Functions
Abstract Classes and Interfaces

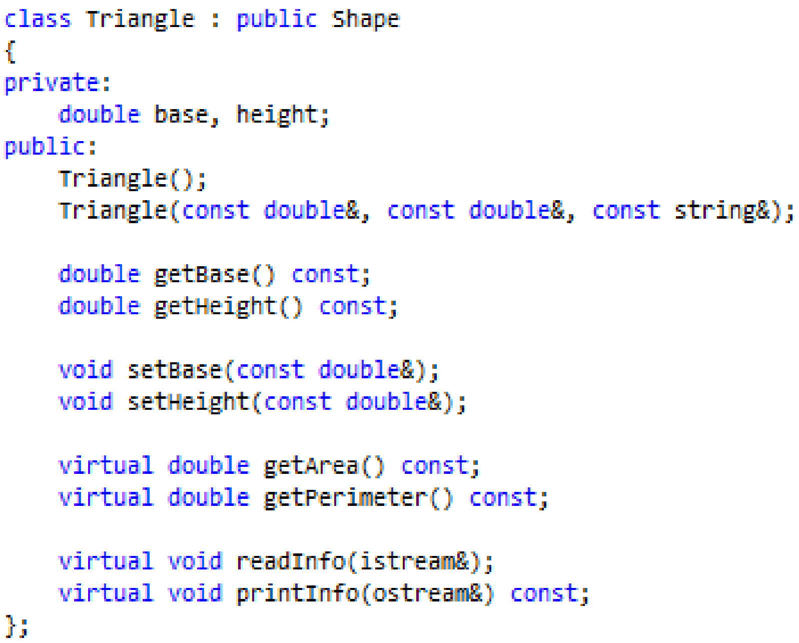


```cpp
class Square : public Rectangle
{
public:
    Square();
    Square(const double&, const string&);

    double getSide() const;

    virtual void setLength(const double&);
    virtual void setWidth(const double&);
    void setSide(const double&);

    virtual void readInfo(istream&);
    virtual void printInfo(ostream&) const;
};
```

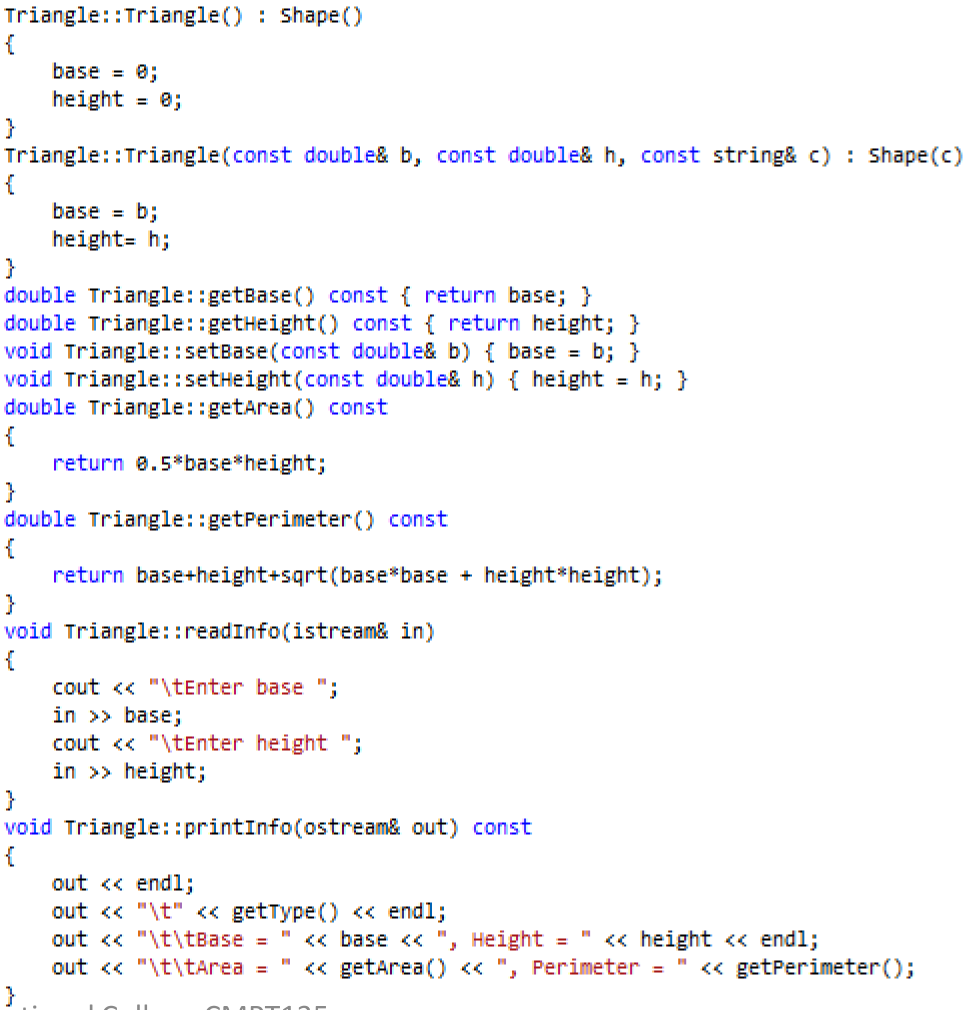


```cpp
Square::Square() : Rectangle() { }
Square::Square(const double& side, const string& c) : Rectangle(side, side, c) { }
double Square::getSide() const
{
    return getLength();
}
void Square::setLength(const double& len)
{
    setSide(len);
}
void Square::setWidth(const double& wid)
{
    setSide(wid);
}
void Square::setSide(const double& side)
{
    this->Rectangle::setLength(side);
    this->Rectangle::setWidth(side);
}
void Square::readInfo(istream& in)
{
    double temp;
    cout << "\tEnter side ";
    in >> temp;
    this->setSide(temp);
}
void Square::printInfo(ostream& out) const
{
    out << endl;
    out << "\t" << getType() << endl;
    out << "\t\tSide = " << getSide() << endl;
    out << "\t\tArea = " << getArea() << ", Perimeter = " << getPerimeter();
}
```

Pure Virtual Functions
Abstract Classes and Interfaces

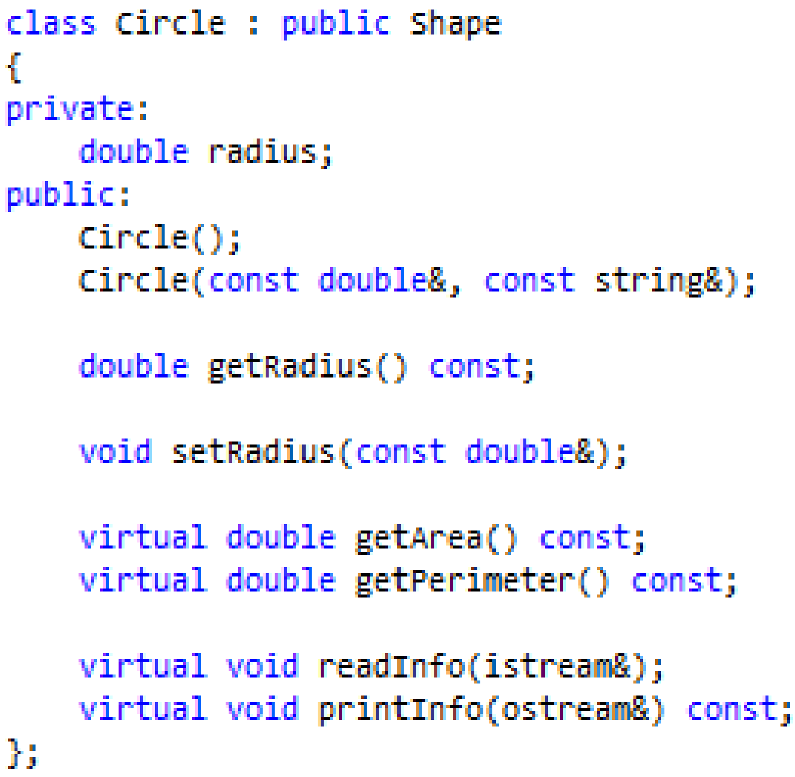


```cpp
class Triangle : public Shape
{
private:
    double base, height;
public:
    Triangle();
    Triangle(const double&, const double&, const string&);

    double getBase() const;
    double getHeight() const;

    void setBase(const double&);
    void setHeight(const double&);

    virtual double getArea() const;
    virtual double getPerimeter() const;

    virtual void readInfo(istream&);
    virtual void printInfo(ostream&) const;
};
```

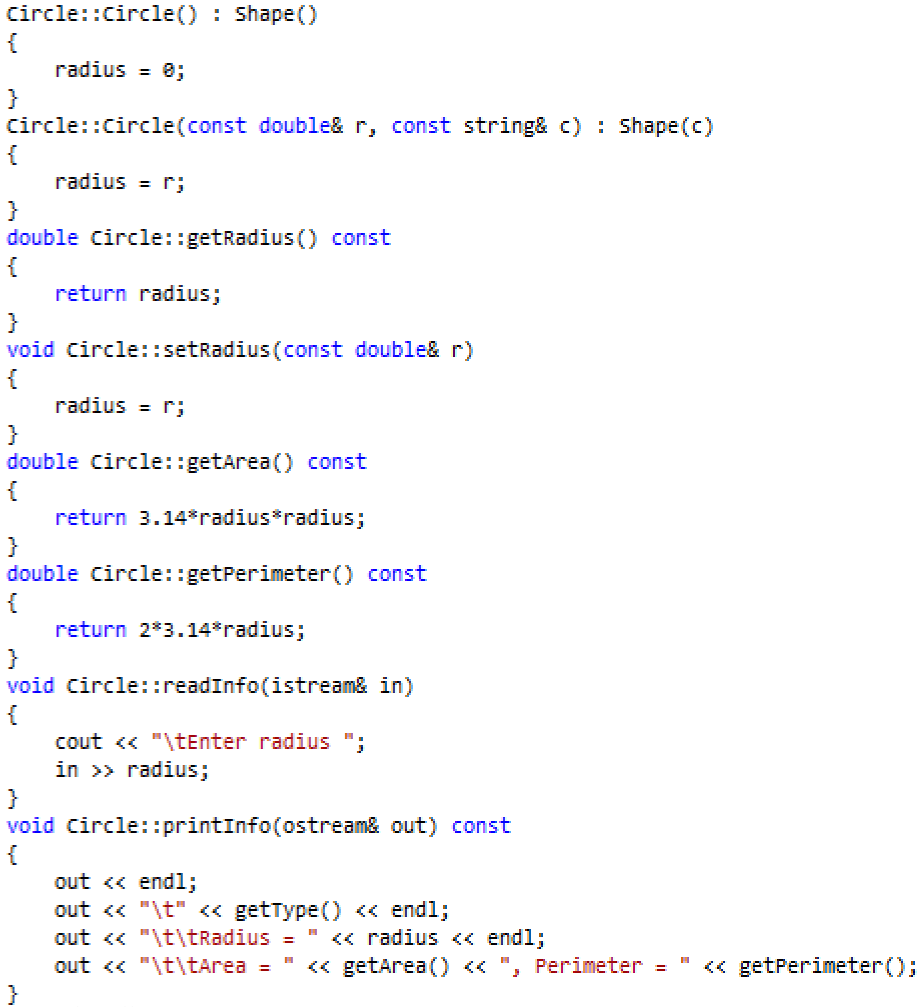


```cpp
Triangle::Triangle() : Shape()
{
    base = 0;
    height = 0;
}
Triangle::Triangle(const double& b, const double& h, const string& c) : Shape(c)
{
    base = b;
    height= h;
}
double Triangle::getBase() const { return base; }
double Triangle::getHeight() const { return height; }
void Triangle::setBase(const double& b) { base = b; }
void Triangle::setHeight(const double& h) { height = h; }
double Triangle::getArea() const
{
    return 0.5*base*height;
}
double Triangle::getPerimeter() const
{
    return base+height+sqrt(base*base + height*height);
}
void Triangle::readInfo(istream& in)
{
    cout << "\tEnter base ";
    in >> base;
    cout << "\tEnter height ";
    in >> height;
}
void Triangle::printInfo(ostream& out) const
{
    out << endl;
    out << "\t" << getType() << endl;
    out << "\t\tBase = " << base << ", Height = " << height << endl;
    out << "\t\tArea = " << getArea() << ", Perimeter = " << getPerimeter();
}
```

Pure Virtual Functions Abstract Classes and Interfaces



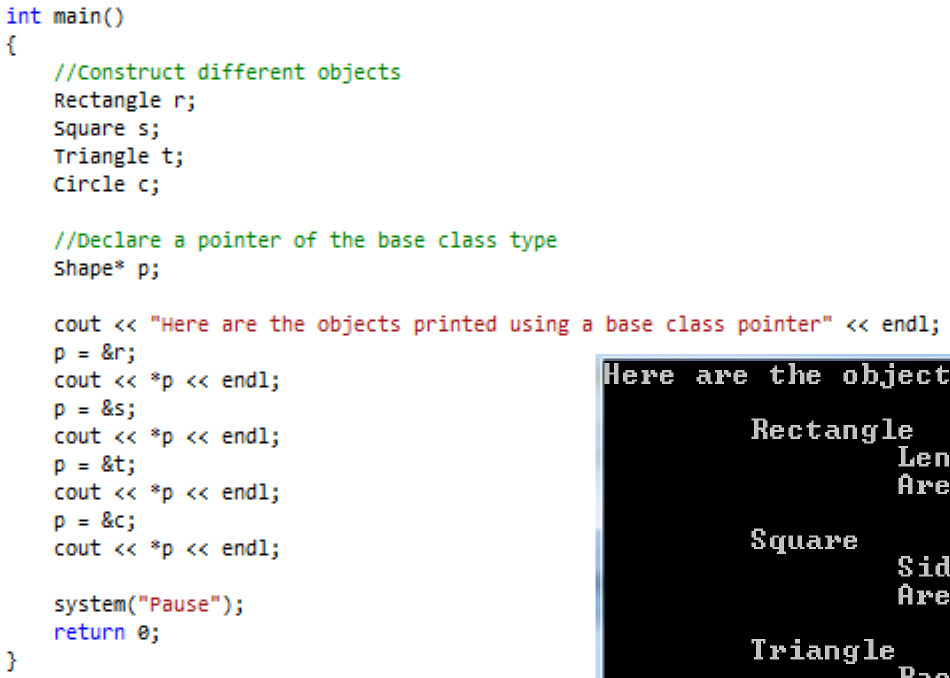
```
class Circle : public Shape
{
private:
    double radius;
public:
    Circle();
    Circle(const double&, const string&);

    double getRadius() const;

    void setRadius(const double&);

    virtual double getArea() const;
    virtual double getPerimeter() const;

    virtual void readInfo(istream&);
    virtual void printInfo(ostream&) const;
};
```



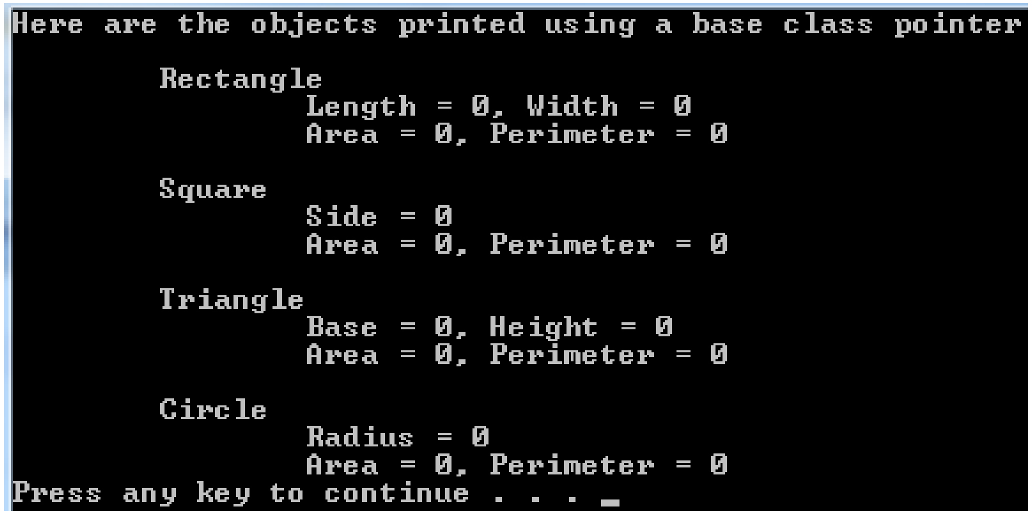
```
Circle::Circle() : Shape()
{
    radius = 0;
}
Circle::Circle(const double& r, const string& c) : Shape(c)
{
    radius = r;
}
double Circle::getRadius() const
{
    return radius;
}
void Circle::setRadius(const double& r)
{
    radius = r;
}
double Circle::getArea() const
{
    return 3.14*radius*radius;
}
double Circle::getPerimeter() const
{
    return 2*3.14*radius;
}
void Circle::readInfo(istream& in)
{
    cout << "\tEnter radius ";
    in >> radius;
}
void Circle::printInfo(ostream& out) const
{
    out << endl;
    out << "\t" << getType() << endl;
    out << "\t\tRadius = " << radius << endl;
    out << "\t\tArea = " << getArea() << ", Perimeter = " << getPerimeter();
}
```

Pure Virtual Functions
Abstract Classes and Interfaces

- Here is a test program together with its output in order to show the effect of our design



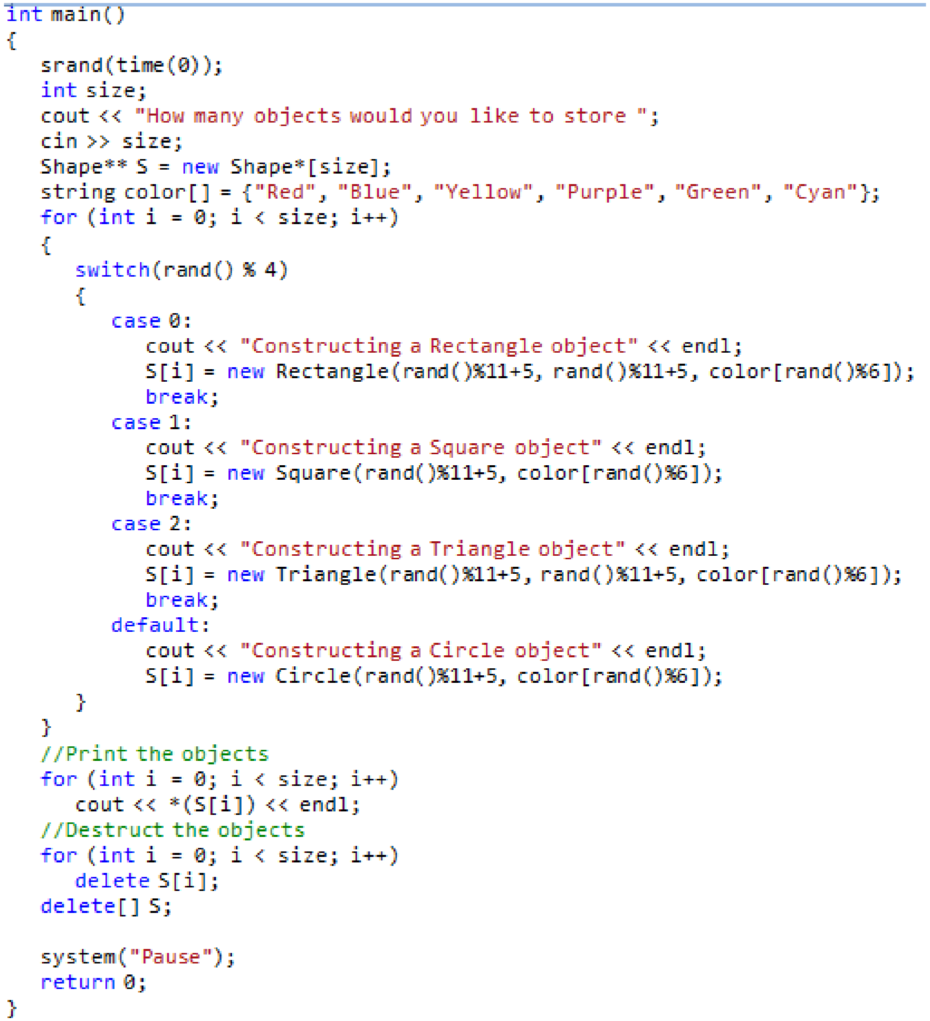
```
int main()
{
    //Construct different objects
    Rectangle r;
    Square s;
    Triangle t;
    Circle c;

    //Declare a pointer of the base class type
    Shape* p;

    cout << "Here are the objects printed using a base class pointer" << endl;
    p = &r;
    cout << *p << endl;
    p = &s;
    cout << *p << endl;
    p = &t;
    cout << *p << endl;
    p = &c;
    cout << *p << endl;

    system("Pause");
    return 0;
}
```



```
Here are the objects printed using a base class pointer

        Rectangle
                Length = 0, Width = 0
                Area = 0, Perimeter = 0

        Square
                Side = 0
                Area = 0, Perimeter = 0

        Triangle
                Base = 0, Height = 0
                Area = 0, Perimeter = 0

        Circle
                Radius = 0
                Area = 0, Perimeter = 0
Press any key to continue . . . _
```

Pure Virtual Functions Abstract Classes and Interfaces

- Now we may create an array of Shape pointers and point the elements of the array to different objects
- Traversing the elements of the array and invoking some member functions will then invoke the correct member function of the underlying objects thanks to the polymorphic behavior of member functions achieved with the help of virtual functions



```
int main()
{
    srand(time(0));
    int size;
    cout << "How many objects would you like to store ";
    cin >> size;
    Shape** S = new Shape*[size];
    string color[] = {"Red", "Blue", "Yellow", "Purple", "Green", "Cyan"};
    for (int i = 0; i < size; i++)
    {
        switch(rand() % 4)
        {
            case 0:
                cout << "Constructing a Rectangle object" << endl;
                S[i] = new Rectangle(rand()%11+5, rand()%11+5, color[rand()%6]);
                break;
            case 1:
                cout << "Constructing a Square object" << endl;
                S[i] = new Square(rand()%11+5, color[rand()%6]);
                break;
            case 2:
                cout << "Constructing a Triangle object" << endl;
                S[i] = new Triangle(rand()%11+5, rand()%11+5, color[rand()%6]);
                break;
            default:
                cout << "Constructing a Circle object" << endl;
                S[i] = new Circle(rand()%11+5, color[rand()%6]);
        }
    }
    //Print the objects
    for (int i = 0; i < size; i++)
        cout << *(S[i]) << endl;
    //Destruct the objects
    for (int i = 0; i < size; i++)
        delete S[i];
    delete[] S;

    system("Pause");
    return 0;
}
```