# Inheritance in C++

**Fundamental Computer Programming- C++ Lab (II)**


元智大學 資訊工程學系
Department of Computer Science & Engineering

*Lecturer:* ***Ho Quang Thai***

# Outline

- Introduction
- Base and Derived Classes
- Access Control and Interitance
- Type of Inheritance
- Multiple Inheritance
- Exercises

# Introduction

- One of the most important concepts in object-oriented programming.
- Allows us to define a class in terms of another class.
- Makes it easier to create and maintain an application.
- Provides an opportunity to reuse the code functionality and fast implementation time.

# Introduction

- When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class.
- This existing class is called the **base** class, and the new class is referred to as the **derived** class.
- The idea of inheritance implements the **is a relationship**.

# Base and Derived Classes

- A class can be derived from more than one classes
- It means it can inherit data and functions from multiple base classes.
- To define a derived class, we use a class derivation list to specify the base class(es).
- A class derivation list names one or more base classes and has the form.

```
class derived-class: access-specifier base-class
```

# Example

Base class

Derived class

```cpp
#include <iostream>

using namespace std;

// Base class
class Shape {
   public:
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Derived class
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};

int main(void) {
   Rectangle Rect;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

OUTPUT

```
Total area: 35
```

# Access Control and Inheritance

- A derived class can access all the non-private members of its base class.
- Summarize the different access types

| Access | public | protected | private |
|---|---|---|---|
| **Same class** | yes | yes | yes |
| **Derived classes** | yes | yes | no |
| **Outside classes** | yes | no | no |

# Access Control and Inheritance

- A derived class inherits all base class methods with the following **exceptions:**
  - Constructors, destructors and copy constructors of the base class.
  - Overloaded operators of the base class.
  - The friend functions of the base class.

# Type of Inheritance

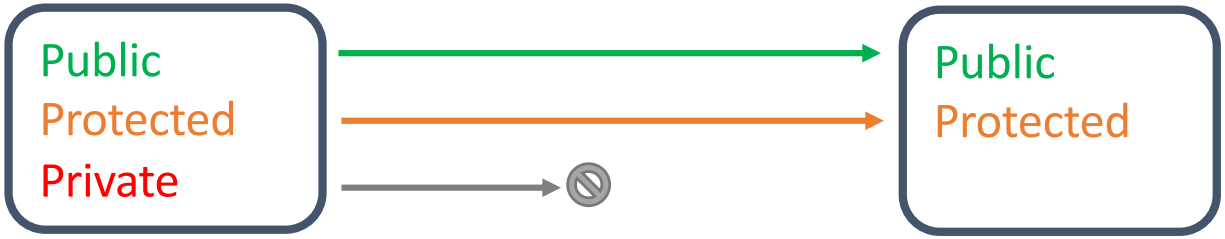
- When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance.

```
class derived-class: access-specifier base-class
```

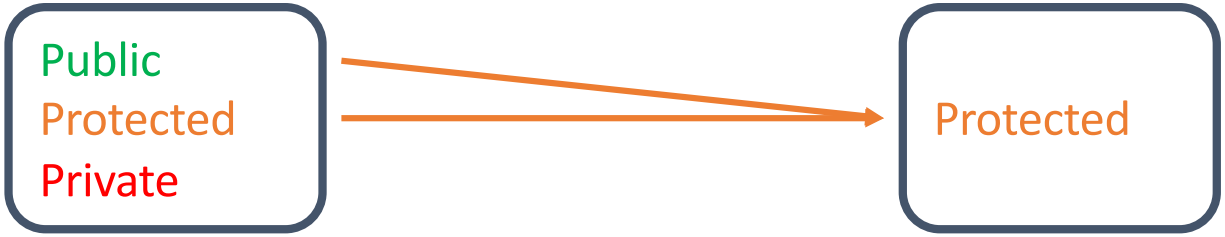
- We hardly use protected or private inheritance, but public inheritance is commonly used

# Type of Inheritance

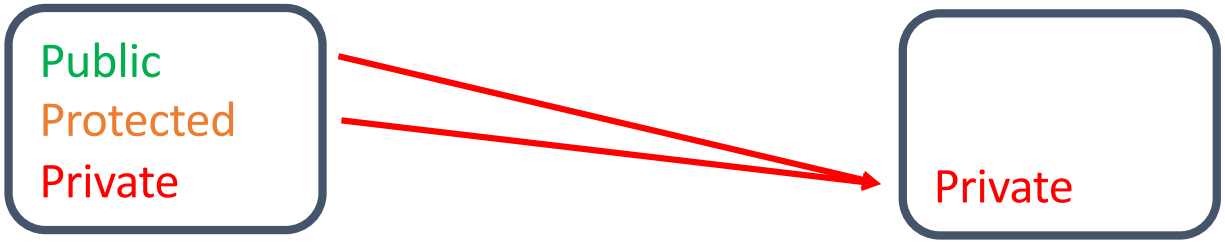
**Public Inheritance**



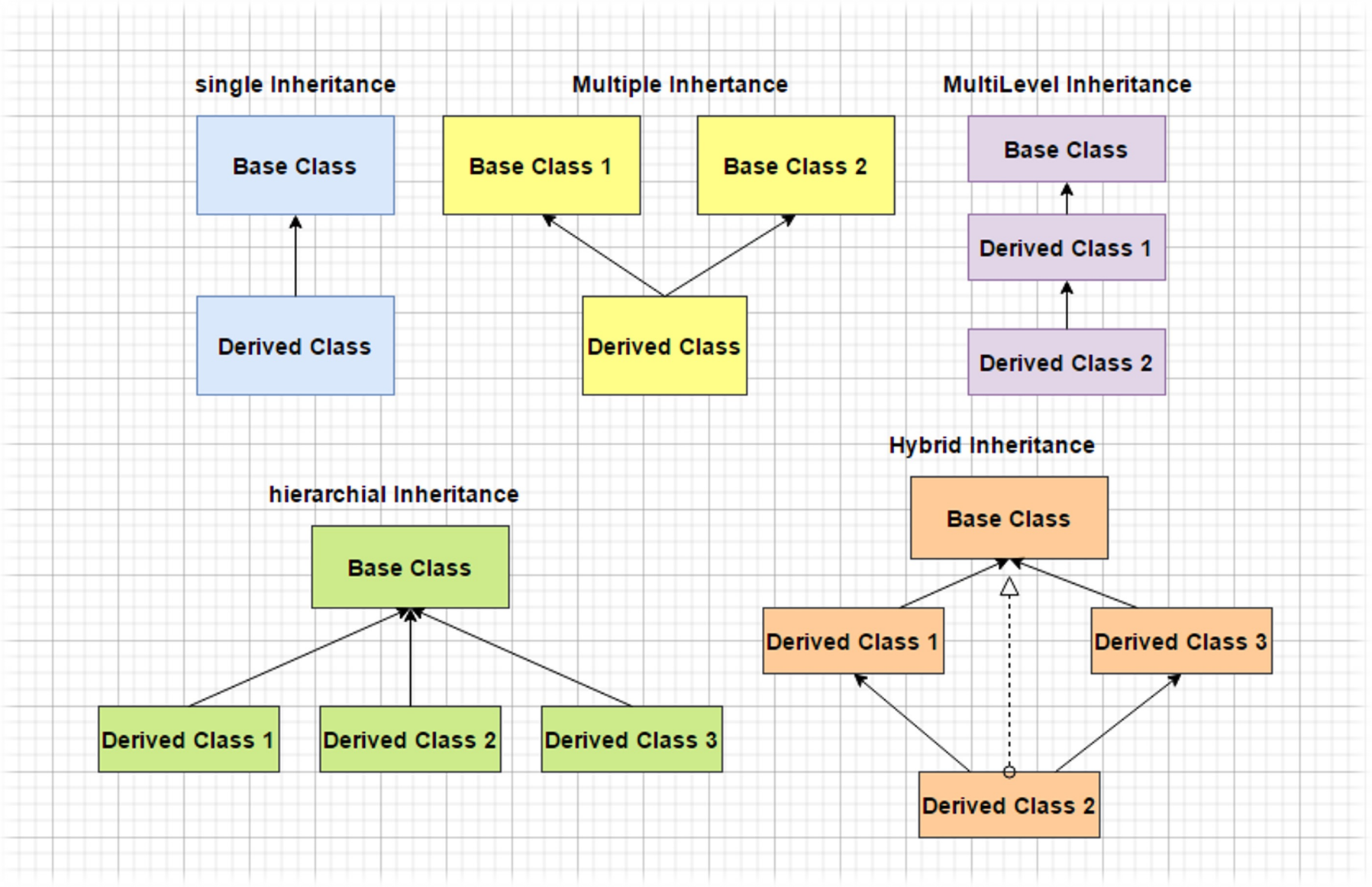

**Protected Inheritance**




**Private Inheritance**

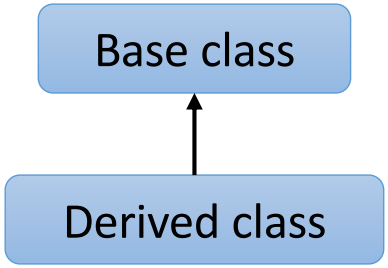


# Type of Inheritance



# Type of Inheritance

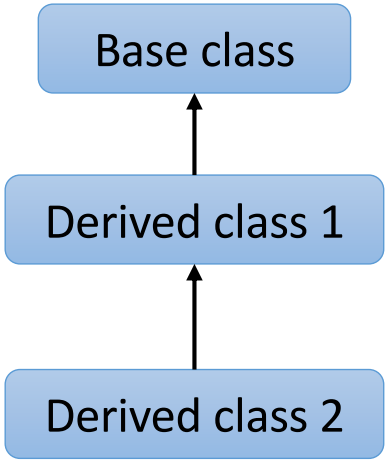
## Single Interitance




```cpp
#include <iostream>

class Animal {
public:
    void eat() {
        std::cout << "Animal is eating" << std::endl;
    }
};

class Dog : public Animal {
public:
    void bark() {
        std::cout << "Dog is barking" << std::endl;
    }
};

int main() {
    Dog myDog;
    myDog.eat();  // Inherited from class Animal
    myDog.bark();
    return 0;
}
```

# Type of Inheritance

## Multi-level Interitance




```cpp
#include <iostream>

class Parent {
public:
    void parentFunction() {
        std::cout << "Parent function" << std::endl;
    }
};

class Child : public Parent {
public:
    void childFunction() {
        std::cout << "Child function" << std::endl;
    }
};

class GrandChild : public Child {
public:
    void grandChildFunction() {
        std::cout << "Grandchild function" << std::endl;
    }
};

int main() {
    GrandChild myGrandChild;
    myGrandChild.parentFunction();
    myGrandChild.childFunction();
    myGrandChild.grandChildFunction();
    return 0;
}
```

# Type of Inheritance

## Multiple Interitance

- A C++ class can inherit members from more than one class and here is the extended syntax

```
class derived-class: access base-A, access base-B …
```

- Where access is one of public, protected, or private and would be given for every base class and they will be separated by comma as shown above.

# Type of Inheritance

## Multiple Interitance

Base class
**Shape**

Base class
**PaintCost**

Derived class

```cpp
#include <iostream>

using namespace std;

// Base class Shape
class Shape {
   public:
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Base class PaintCost
class PaintCost {
   public:
      int getCost(int area) {
         return area * 70;
      }
};

// Derived class
class Rectangle: public Shape, public PaintCost {
   public:
      int getArea() {
         return (width * height);
      }
};

int main(void) {
   Rectangle Rect;
   int area;

   Rect.setWidth(5);
   Rect.setHeight(7);

   area = Rect.getArea();

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   // Print the total cost of painting
   cout << "Total paint cost: $" << Rect.getCost(area) << endl;

   return 0;
}
```

OUTPUT

```
Total area: 35
Total paint cost: $2450
```

# Type of Inheritance

## Hybrid inheritance

- Hybrid inheritance is used to solve the "diamond problem" in multiclass inheritance.
- This problem occurs when a class inherits from two classes that have the same base class.
- Virtual inheritance ensures that *only one copy of the base class is created*.

```cpp
#include <iostream>

class Component {
public:
    int value;
};

class GraphicsComponent : public virtual Component {
public:
    void draw() {
        std::cout << "Drawing graphics component" << std::endl;
    }
};

class PhysicsComponent : public virtual Component {
public:
    void updatePhysics() {
        std::cout << "Updating physics component" << std::endl;
    }
};

class GameObject : public GraphicsComponent, public PhysicsComponent {
public:
    void update() {
        draw();
        updatePhysics();
    }
};

int main() {
    GameObject obj;
    obj.value = 10; // There is only one copy of the Component class
    obj.update();
    return 0;
}
```

# Method Overriding

- Keyword **virtual** used to declare *virtual method* in base class.
- Keyword **override** used in derived class to indicate that the virtual method *is overriding* a method from the base class.

```cpp
#include <iostream>

class Animal {
public:
    virtual void makeSound() {
        std::cout << "Animal makes a sound" << std::endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() override {
        std::cout << "Dog barks" << std::endl;
    }
};

int main() {
    Animal* myAnimal = new Dog();
    myAnimal->makeSound(); // Outputs "Dog barks"
    delete myAnimal;
    return 0;
}
```

# Covariance

- **Covariance** essentially means that you can use a more derived type (a subtype) in place of a base type (a supertype) in certain contexts, particularly with return types of virtual functions.

```cpp
#include <iostream>
#include <vector>

class Window {
public:
    virtual Window* clone() const {
        std::cout << "Cloning a basic window." << std::endl;
        return new Window(*this);
    }

    virtual void draw() const {
        std::cout << "Drawing a basic window." << std::endl;
    }
};

class TextWindow : public Window {
public:
    TextWindow* clone() const override {
        std::cout << "Cloning a text window." << std::endl;
        return new TextWindow(*this);
    }

    void draw() const override {
        std::cout << "Drawing a text window." << std::endl;
    }
};

int main() {
    std::vector<Window*> windows;
    windows.push_back(new Window());
    windows.push_back(new TextWindow());

    for (Window* w : windows) {
        Window* copy = w->clone();
            // Covariance: TextWindow* when w is TextWindow*
        copy->draw();
        delete copy;
    }

    for (Window* w : windows) {
        delete w;
    }

    return 0;
}
```

# Contravariance

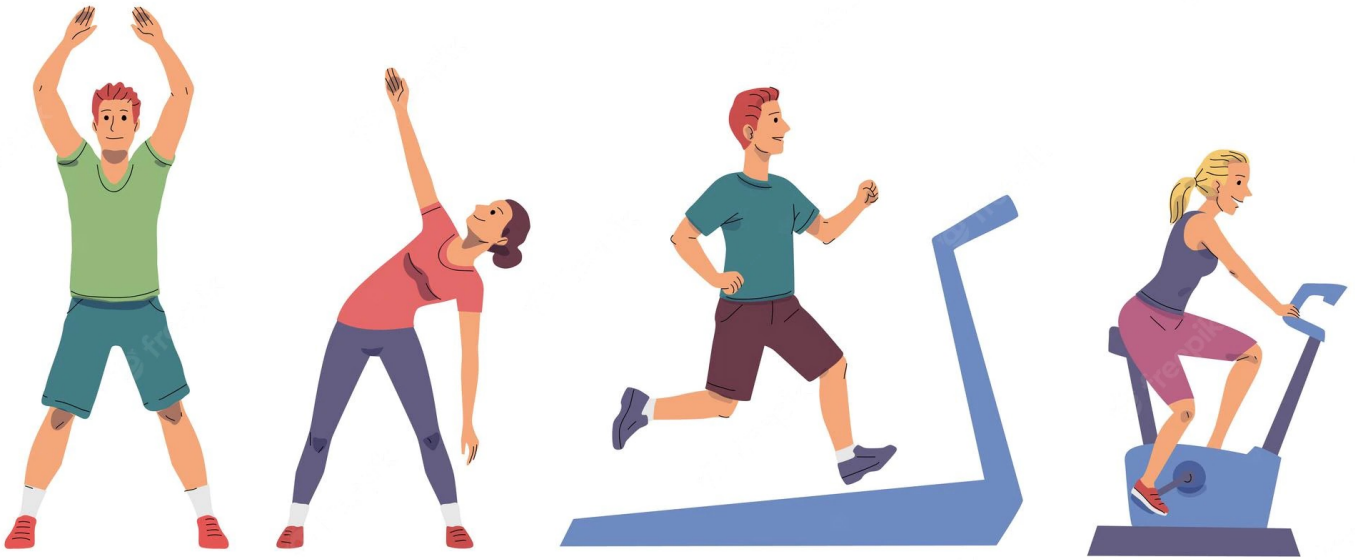
- **Contravariance** is less common in C++ and involves using a more general type (a supertype) in place of a more specific type (a subtype) in certain contexts, typically with function parameters.

```cpp
#include <iostream>

class DataProcessor {
public:
    virtual void process(void* data) {
        std::cout << "Basic data processor." << std::endl;
    }
};

class StringDataProcessor : public DataProcessor {
public:
    void process(void* data) override {
        std::cout << "String data processor." << std::endl;
        // You would typically cast and process the string data here
    }
};

int main() {
    DataProcessor* processor1 = new DataProcessor();
    DataProcessor* processor2 = new StringDataProcessor();

    int number = 10;
    std::string text = "Hello";

    processor1->process(&number);
    processor2->process(&text);
    // Contravariance: StringDataProcessor can handle void*

    delete processor1;
    delete processor2;

    return 0;
}
```

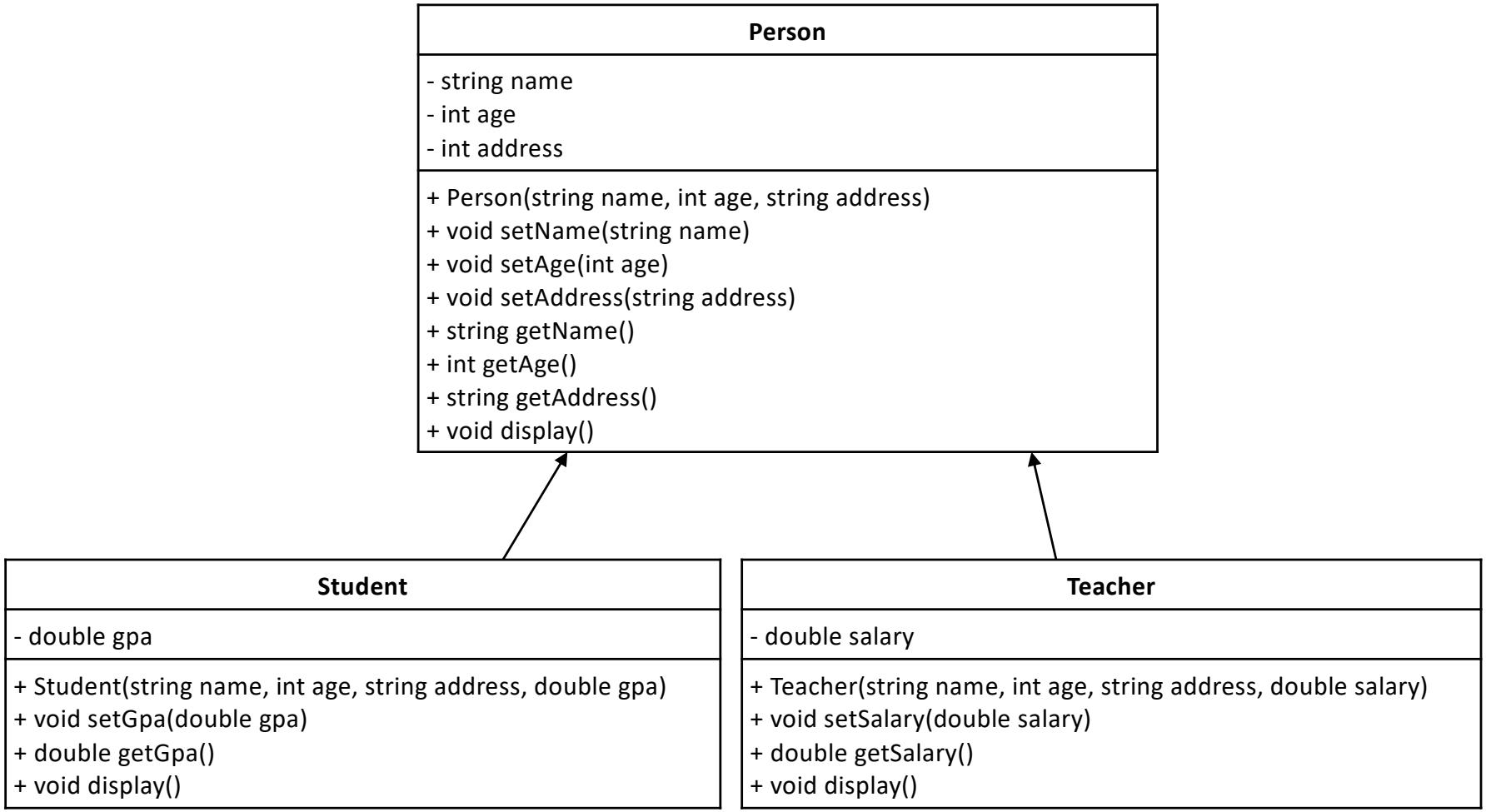
# Contravariance

- **Covariance**: Deals with return types, allowing derived classes to return more specific types.
- **Contravariance**: Deals with parameter types, allowing derived classes to accept more general types.
- **Covariance** is much more common than contravariance in standard C++.
- Careful use of these concepts enhances flexibility and polymorphism in your C++ code.

# Exercises



# Exercise 1


Person
- string name
- int age
- int address
+ Person(string name, int age, string address)
+ void setName(string name)
+ void setAge(int age)
+ void setAddress(string address)
+ string getName()
+ int getAge()
+ string getAddress()
+ void display()
Student
- double gpa
+ Student(string name, int age, string address, double gpa)
+ void setGpa(double gpa)
+ double getGpa()
+ void display()
Teacher
- double salary
+ Teacher(string name, int age, string address, double salary)
+ void setSalary(double salary)
+ double getSalary()
+ void display()

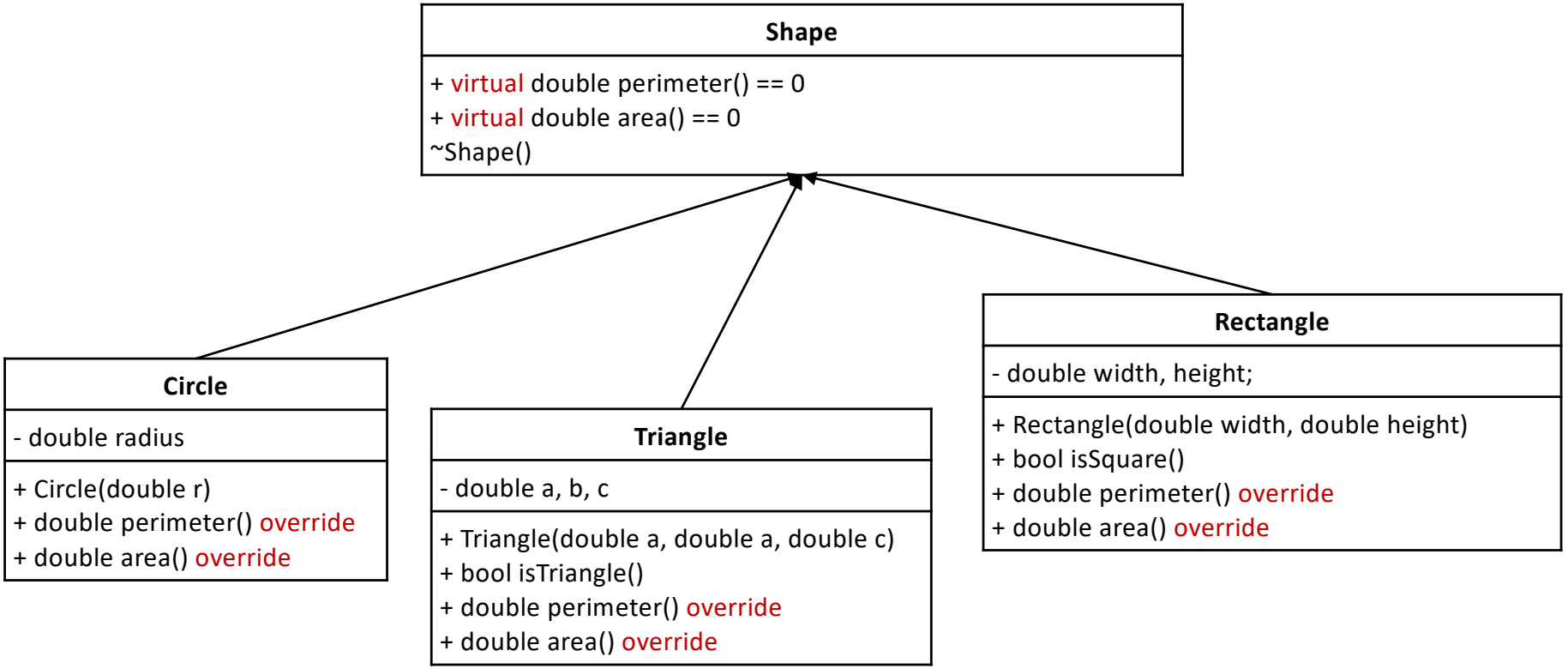
# Exercise 1

Suggested main funtion:

```cpp
#include <iostream>
#include "Person.cpp"
#include "Student.cpp"
#include "Teacher.cpp"

int main() {
    Student s("Lin Jia-Hao",23,"52 Lide Street",9.0);
    s.display();

    Teacher t("Chen Zhu-Wei", 35, "12 ShongShan Road", 42000);
    s.display();

    return 0;
}
```

```
OUTPUT
Name: Lin Jia-Hao
Age: 23
Address: 52 Lide Street
GPA: 9.0

Name: Chen Zhu-Wei
Age: 35
Address: 12 ShongShan Road
Salary: 42,000 TWD
```

# Exercise 2


Shape
+ virtual double perimeter() == 0
+ virtual double area() == 0
~Shape()
Circle
- double radius
+ Circle(double r)
+ double perimeter() override
+ double area() override
Triangle
- double a, b, c
+ Triangle(double a, double a, double c)
+ bool isTriangle()
+ double perimeter() override
+ double area() override
Rectangle
- double width, height;
+ Rectangle(double width, double height)
+ bool isSquare()
+ double perimeter() override
+ double area() override

# Exercise 2

Suggested main funtion:

```
int main() {
    Circle c(5);
    Triangle t(3, 4, 5);
    Rectangle r(4, 6);

    cout << "Perimeter and area of shapes are: " << endl;
    cout << "Circle: " << endl;
    cout << "Perimeter: " << c.perimeter() << endl;
    cout << "Area: " << c.area() << endl;

    cout << "Triangle: " << endl;
    cout << "Perimeter: " << t.perimeter() << endl;
    cout << "Area: " << t.area() << endl;

    cout << "Rectangle: " << endl;
    cout << "Perimeter: " << r.perimeter() << endl;
    cout << "Area: " << r.area() << endl;
    return 0;
}
```

# Questions & Answers