Hochiminh City University of Technology
Computer Science and Engineering
OOP in C++

Advanced OOP

Lecturer: Vu Van Tien

Today's outline

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Template

Features

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Abstraction

- Separating the implementation details from its use
- E.g: (definition, operations) vs (data representation, algorithms inside method implementation)

Encapsulation

- Allows the programmer to group data and the subroutines that operate on them together in one place
- Hide implementation details from the user.
- Data hiding:
  - prevents users from accessing data directly
  - data can be accessed only through the member functions

Inheritance

What Is Inheritance?

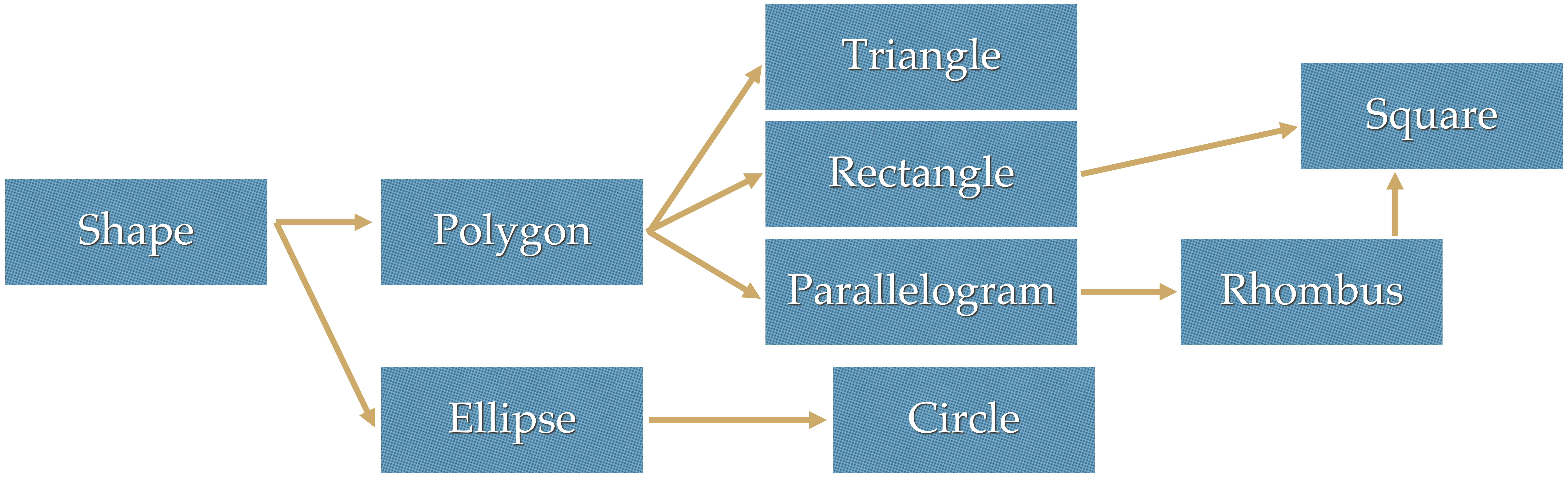
- Provides a way to create a new class from an existing class
- The new class is a specialized version of the existing class

Inheritance – Terminology and Notation in C++

- Base class (or parent) – inherited from
- Derived class (or child) – inherits from the base class
- Notation:

```
class Student           // base class
{
 . . .
};
class UnderGrad : public student
{                  // derived class
 . . .
};
```

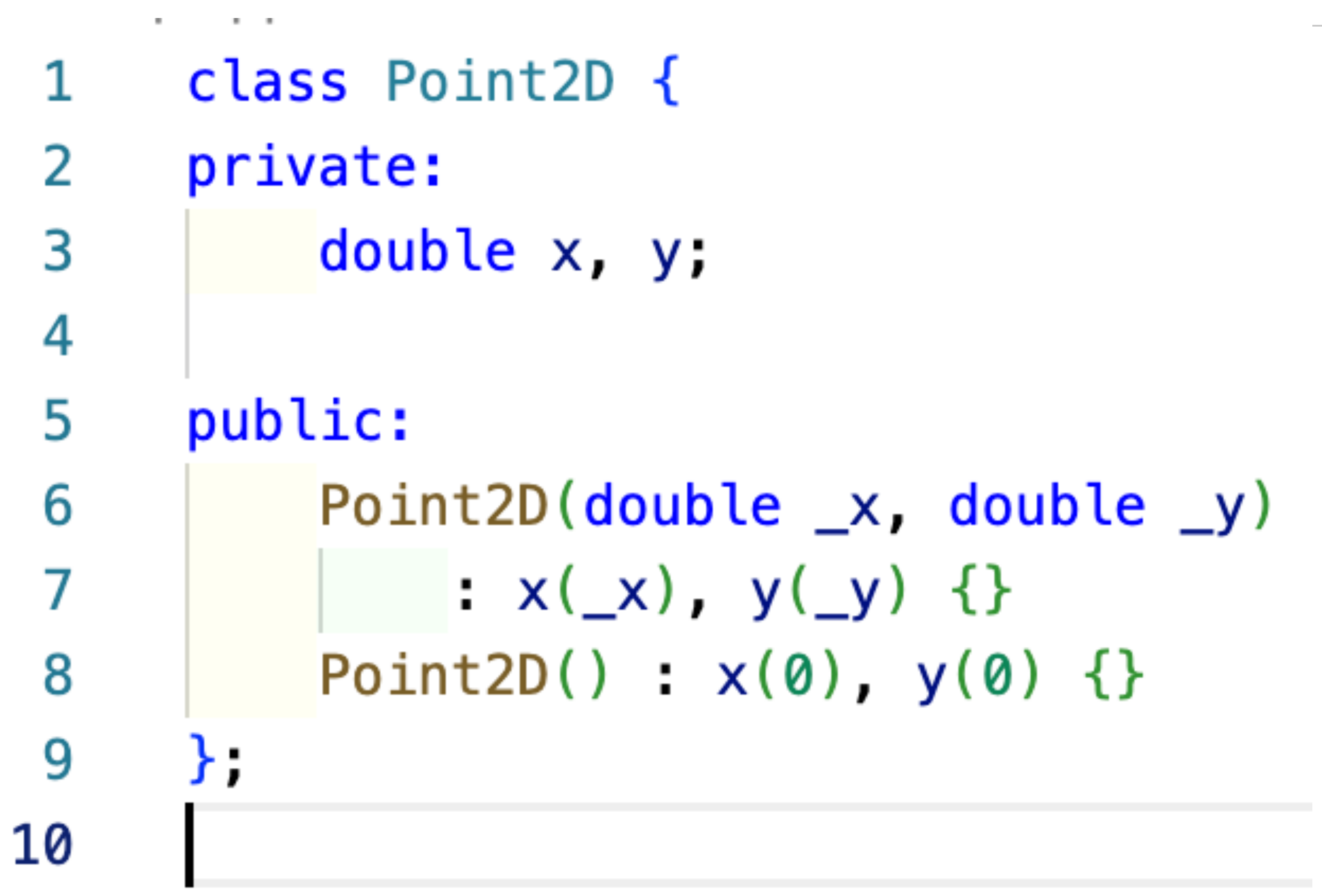
Inheritance Exmaple



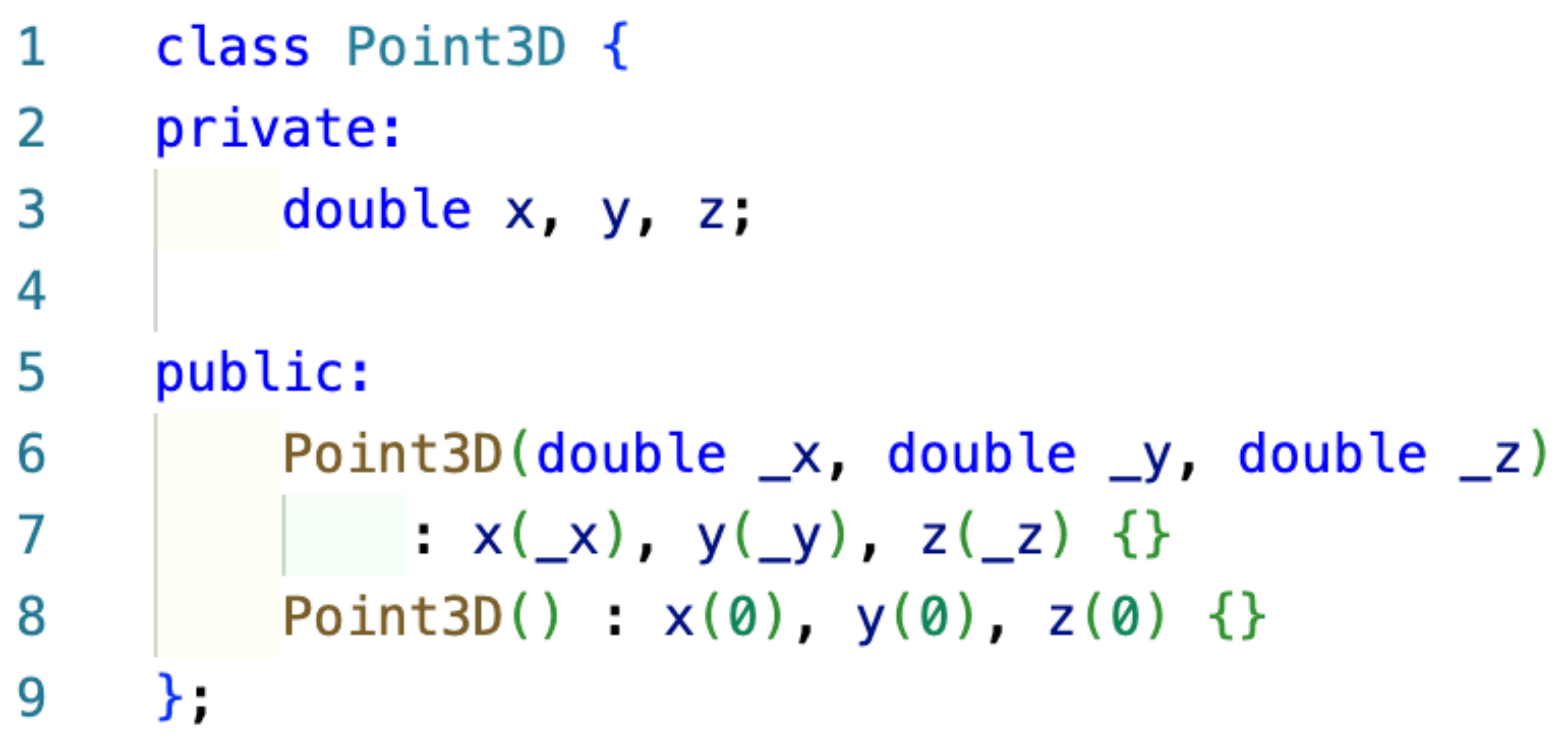
Inheritance Syntax

-
  ```
  class <CName> [: <access specifier> <BaseCName>] {
      . . .
  };
  ```
- E.g.:
  ```
  class Polygon : public Shape { . . . };
  class Rectangle : public Polygon { . . . };
  class Square : public Rectangle, public Rhombus { . . . };
  class Ellipse: public Shape { . . . };
  ```

Example: Point2D & Point3D

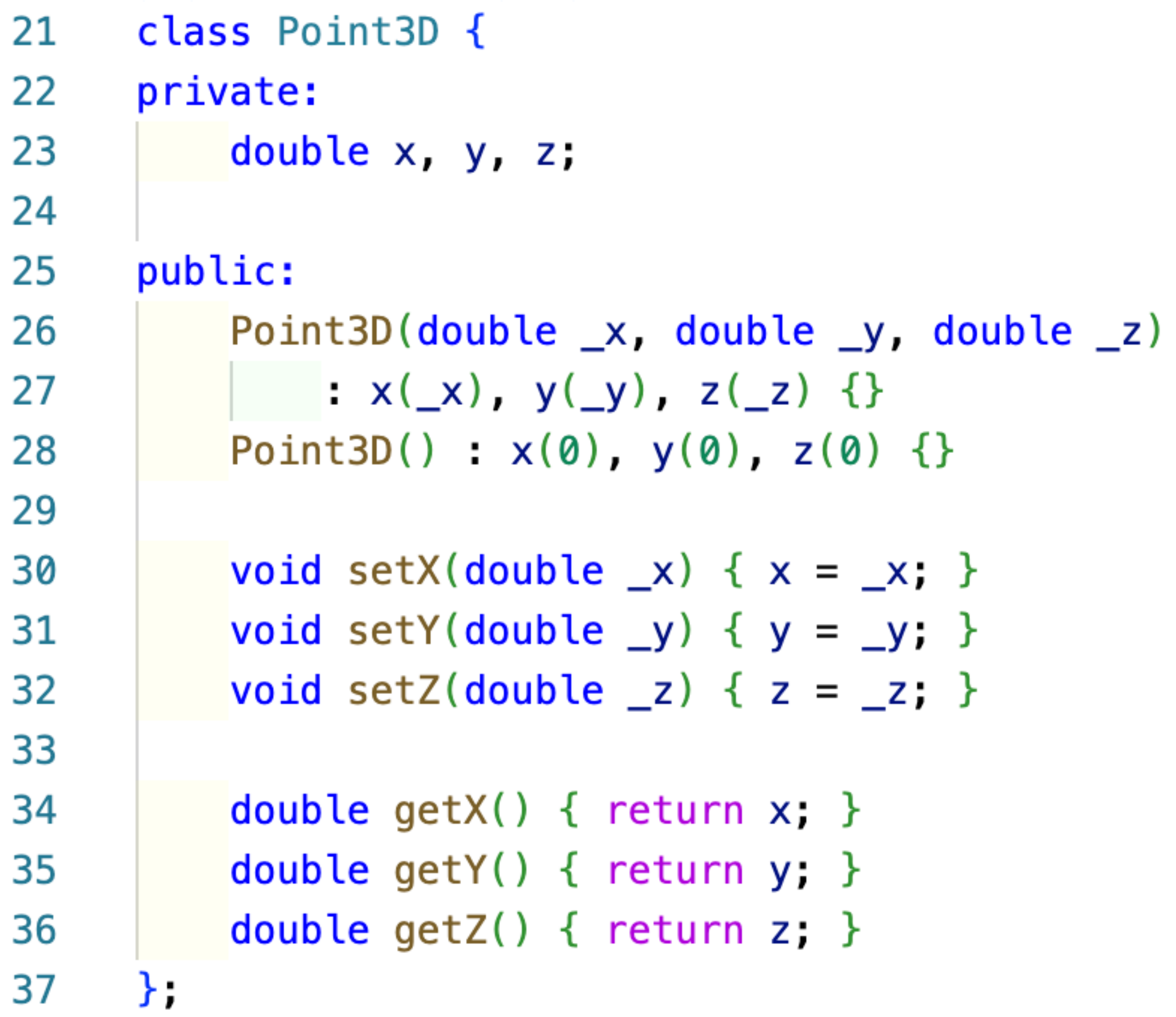


```cpp
class Point2D {
private:
    double x, y;

public:
    Point2D(double _x, double _y)
        : x(_x), y(_y) {}
    Point2D() : x(0), y(0) {}
};
```

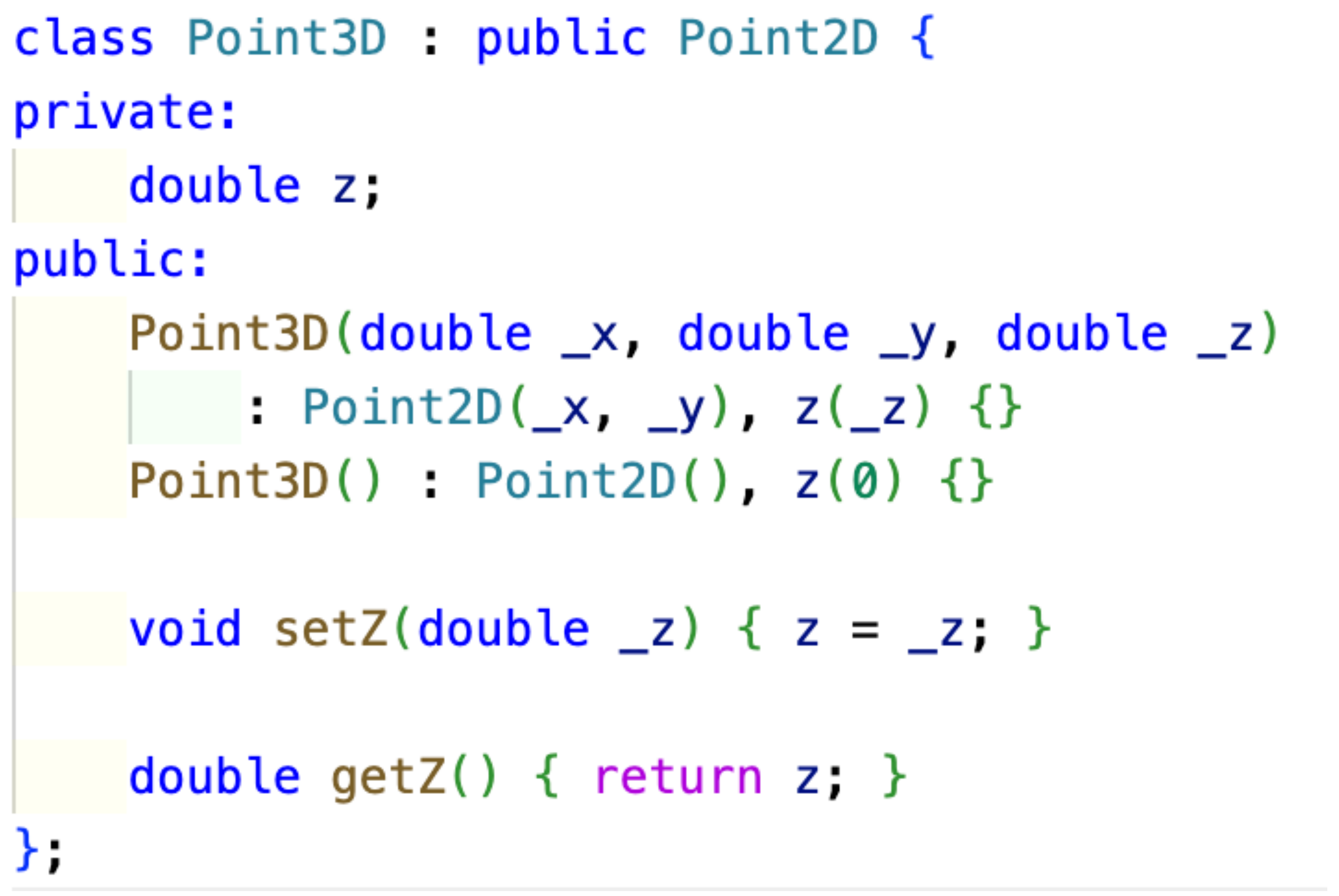


```cpp
class Point3D {
private:
    double x, y, z;

public:
    Point3D(double _x, double _y, double _z)
        : x(_x), y(_y), z(_z) {}
    Point3D() : x(0), y(0), z(0) {}
};
```

Example: Point2D & Point3D



```cpp
class Point3D {
private:
    double x, y, z;

public:
    Point3D(double _x, double _y, double _z)
        : x(_x), y(_y), z(_z) {}
    Point3D() : x(0), y(0), z(0) {}

    void setX(double _x) { x = _x; }
    void setY(double _y) { y = _y; }
    void setZ(double _z) { z = _z; }

    double getX() { return x; }
    double getY() { return y; }
    double getZ() { return z; }
};
```



```cpp
class Point3D : public Point2D {
private:
    double z;
public:
    Point3D(double _x, double _y, double _z)
        : Point2D(_x, _y), z(_z) {}
    Point3D() : Point2D(), z(0) {}

    void setZ(double _z) { z = _z; }

    double getZ() { return z; }
};
```

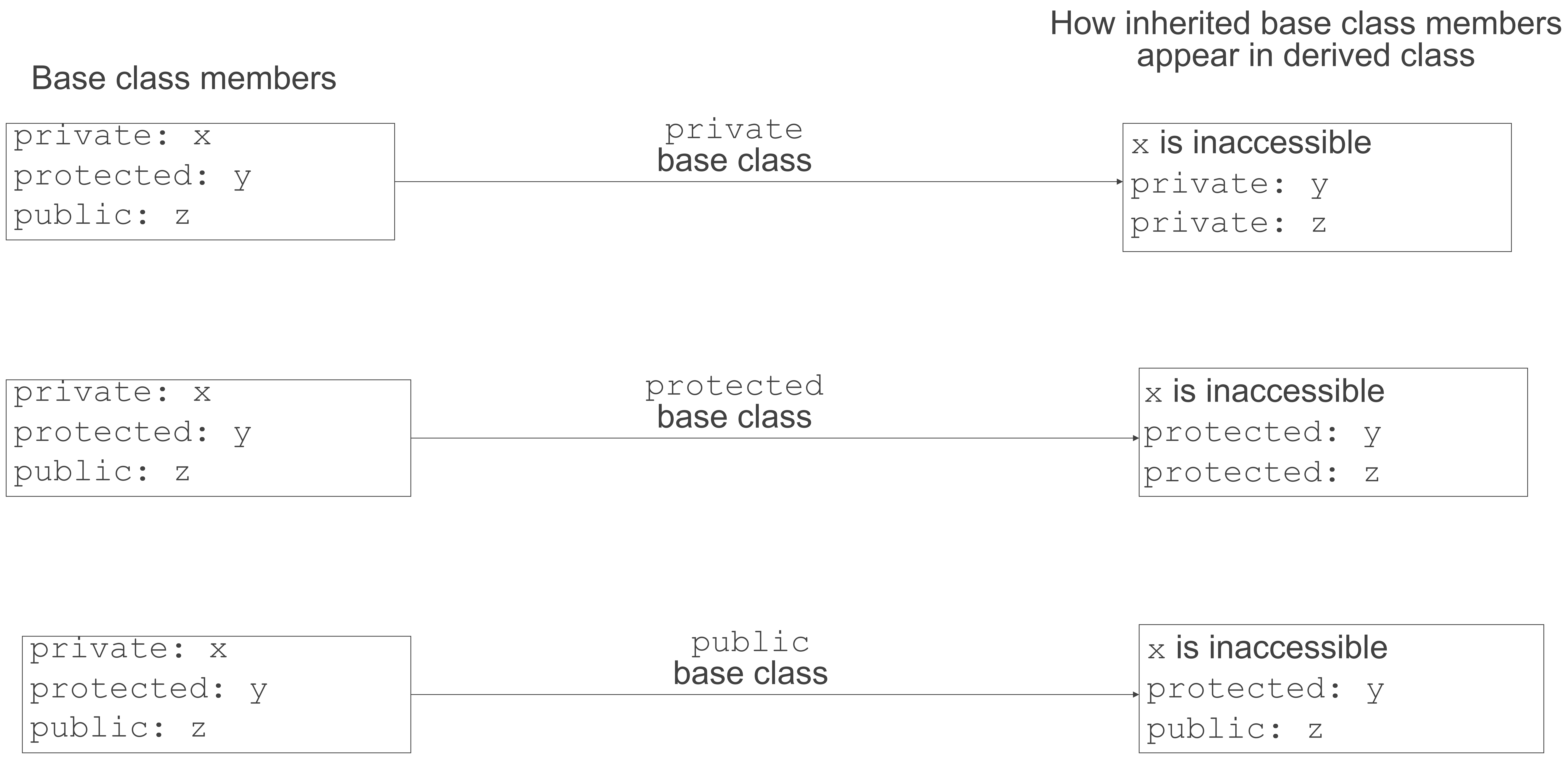
Inheritance's Issues

- Access Specifier
- Overriding Member Functions of Base class
- Constructor & Destructor of Derived & Base class

Protected Access

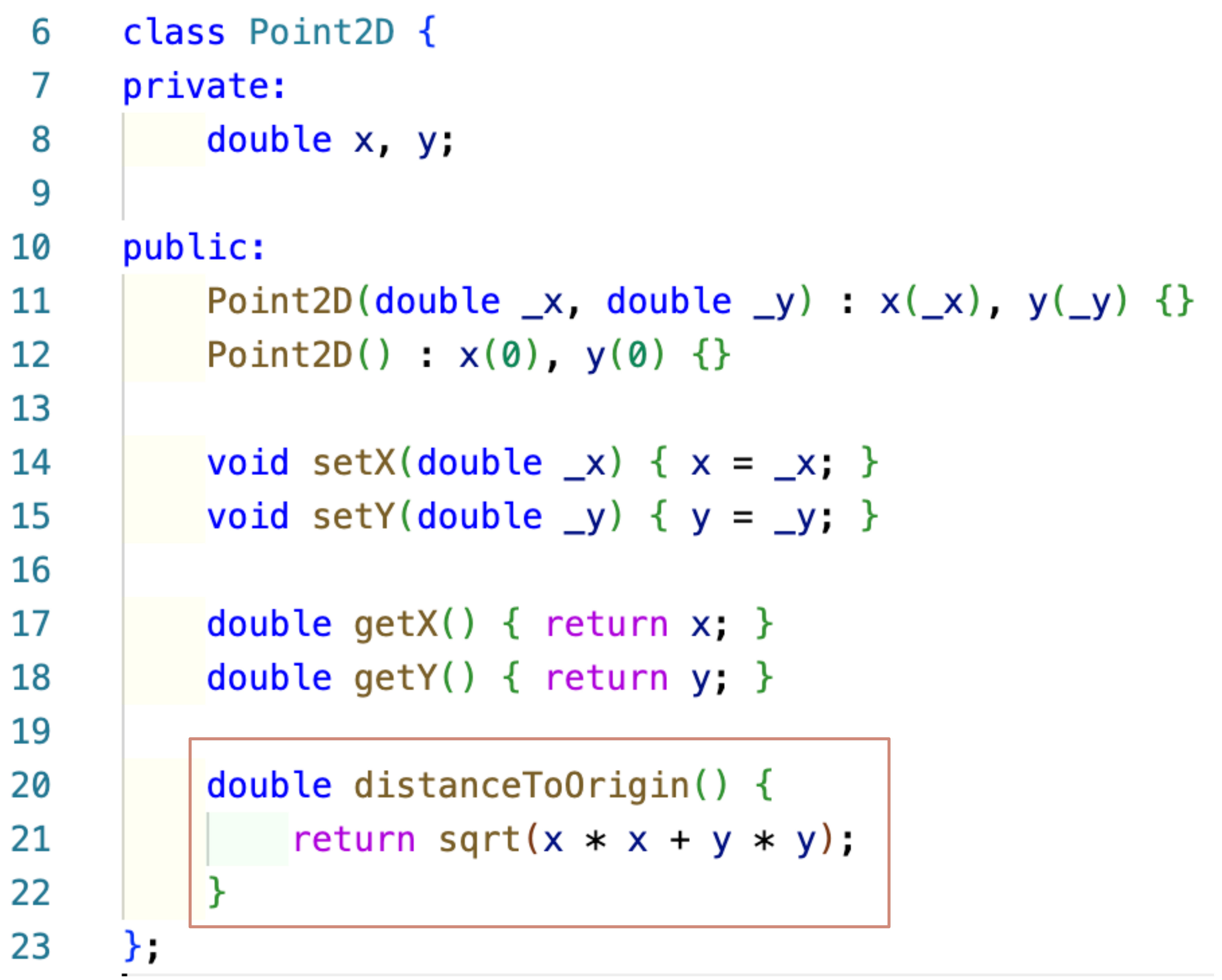
- A base class's **public** members are accessible within its body and anywhere that the program has a handle to an object of that class or one of its derived classes.
- A base class's **private** members are accessible only within its body and to the friends of that base class.
- A base class's **protected** members can be accessed within the body of that base class, by members and friends of that base class, and by members and friends of any classes derived from that base class.
- Using **protected** access offers an **intermediate level** of protection between **public** and **private** access.

Inheritance vs. Access

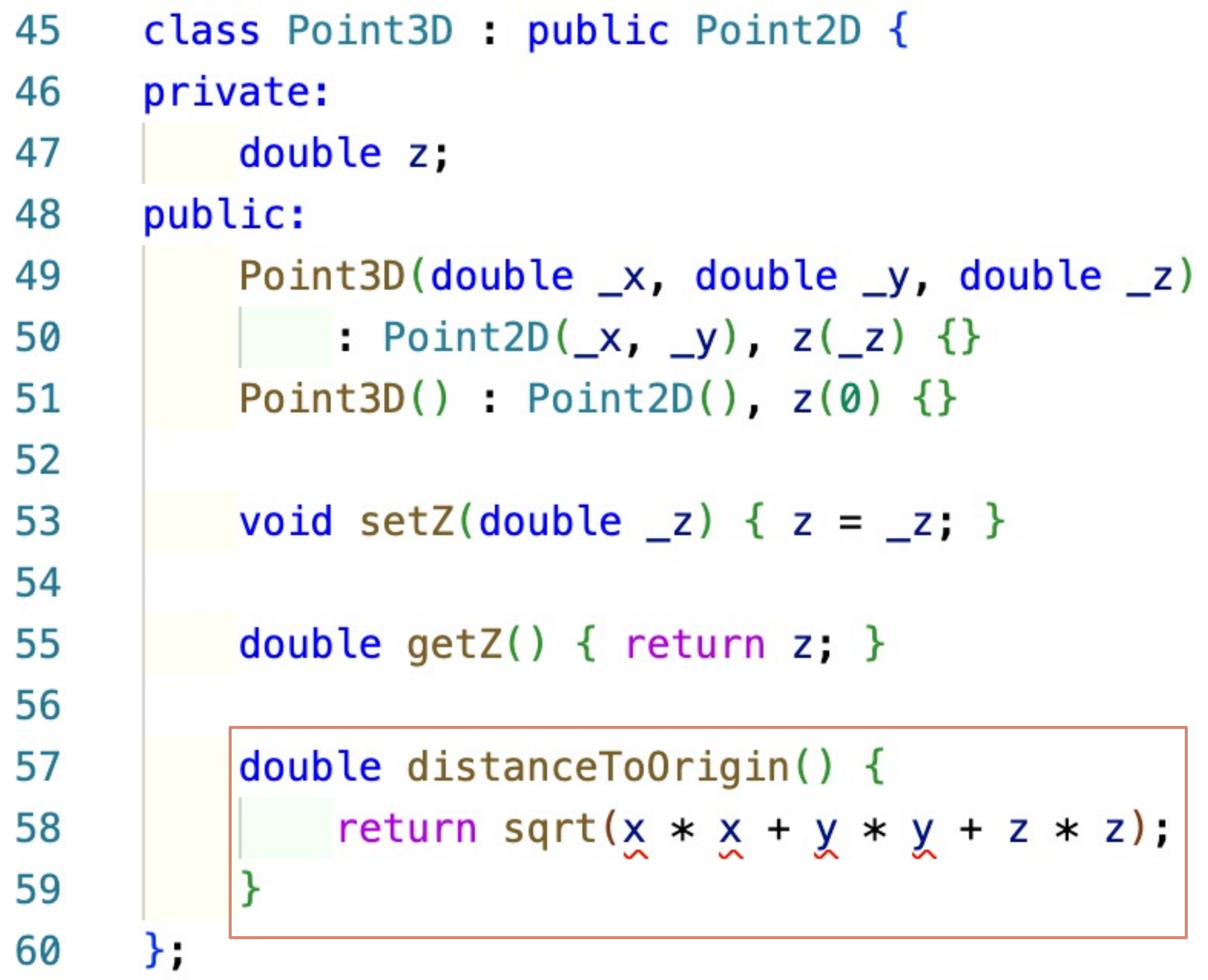


Overriding Member Functions of Base class

- *Derived Class* has its own logic method with the same name with *Base Class*



```cpp
class Point2D {
private:
    double x, y;

public:
    Point2D(double _x, double _y) : x(_x), y(_y) {}
    Point2D() : x(0), y(0) {}

    void setX(double _x) { x = _x; }
    void setY(double _y) { y = _y; }

    double getX() { return x; }
    double getY() { return y; }

    double distanceToOrigin() {
        return sqrt(x * x + y * y);
    }
};
```



```cpp
class Point3D : public Point2D {
private:
    double z;
public:
    Point3D(double _x, double _y, double _z)
        : Point2D(_x, _y), z(_z) {}
    Point3D() : Point2D(), z(0) {}

    void setZ(double _z) { z = _z; }

    double getZ() { return z; }

    double distanceToOrigin() {
        return sqrt(x * x + y * y + z * z);
    }
};
```

Constructors and Destructors in Base and Derived Classes

- Derived classes can have their own constructors and destructors
- When an object of a derived class is created, the base class's constructor is executed first, followed by the derived class's constructor
- When an object of a derived class is destroyed, its destructor is called first, then that of the base class

Passing Arguments to Base Class Constructor

- Allows selection between multiple base class constructors
- Specify arguments to base constructor on derived constructor heading:
  - E.g: Square::Square(int side) : Rectangle(side, side)
- Can also be done with inline constructors
- Must be done if base class has no default constructor

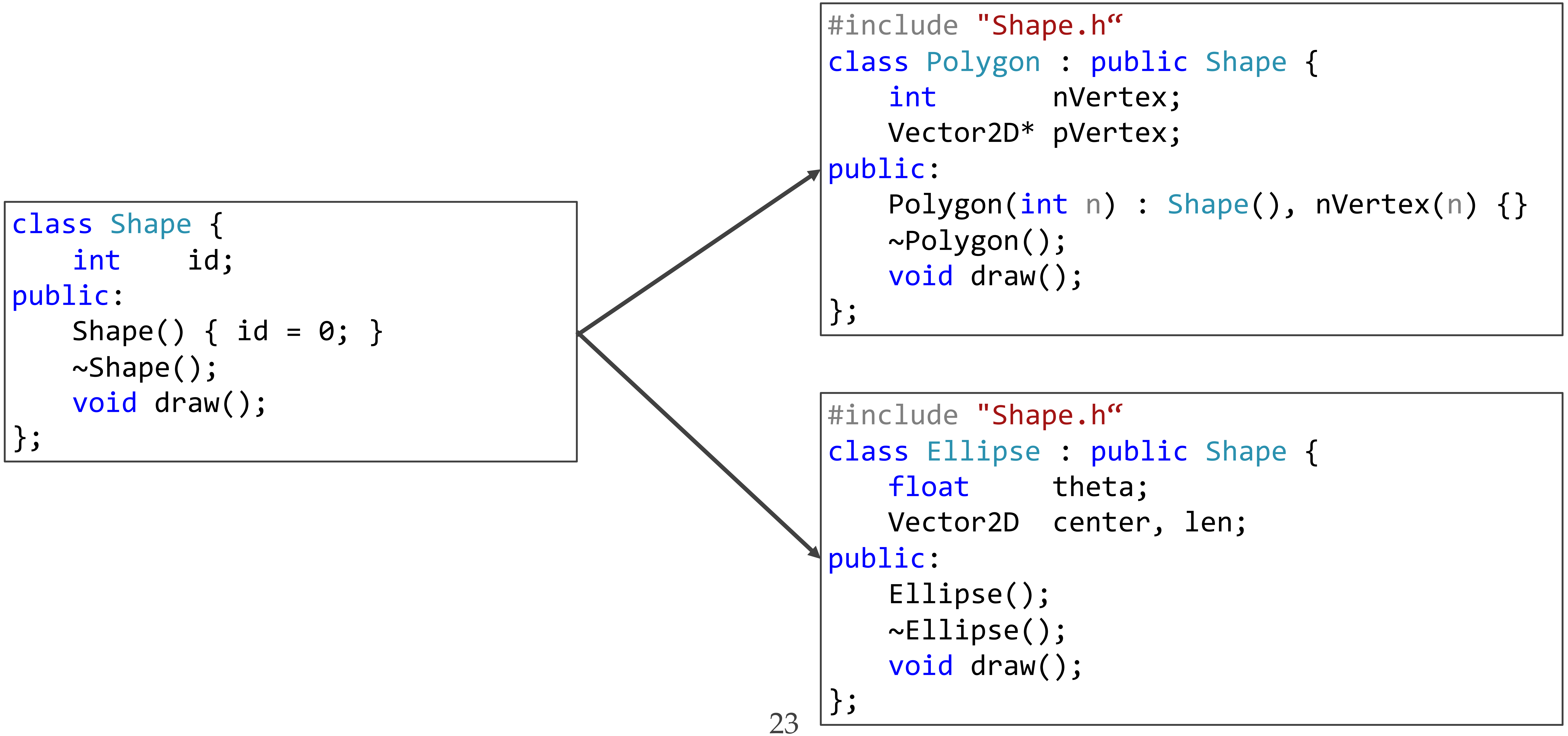
Advantages of inheritance

- When a class inherits from another class, there are three benefits:
  - You can reuse the methods and data of the existing class
  - You can extend the existing class by adding new data and new methods
  - You can modify the existing class by overloading its methods with your own implementations

The "is a" Relationship

- Inheritance establishes an "is a" relationship between classes.
  - A poodle is a dog
  - A car is a vehicle
  - A flower is a plant
  - A football player is an athlete

Inheritance Example



```cpp
class Shape {
    int     id;
public:
    Shape() { id = 0; }
    ~Shape();
    void draw();
};
```

```cpp
#include "Shape.h“
class Polygon : public Shape {
    int        nVertex;
    Vector2D* pVertex;
public:
    Polygon(int n) : Shape(), nVertex(n) {}
    ~Polygon();
    void draw();
};
```

```cpp
#include "Shape.h“
class Ellipse : public Shape {
    float      theta;
    Vector2D  center, len;
public:
    Ellipse();
    ~Ellipse();
    void draw();
};
```

Back to the ‘is a’ Relationship

- An object of a derived class 'is a(n)' object of the base class
- Example:

  - an `UnderGrad` is a `Student`

  - a `Mammal` is an `Animal`
- A derived object has **all** of the characteristics of the base class

What Does a Child Have?

An object of the derived class has:

- all members defined in child class
- all members declared in parent class

An object of the derived class can use:

- all `public` members defined in child class
- all `public` members defined in parent class

Rules for building a class hierarchy

- Derived classes are special cases of base classes
- A derived class can also serve as a base class for new classes.
- There is no limit on the depth of inheritance allowed in C++ (as far as it is within the limits of your compiler)
- It is possible for a class to be a base class for more than one derived class

Polymophism

Outline

Introduction

- **`virtual`** functions and polymorphism
  - Design and implement systems that are more easily extensible
  - Programs written to generically process objects of all existing classes in a hierarchy

Type Fields and `switch` Statements

- **`switch`** statement
  - –Take an action on a object based on its type
  - –A **`switch`** structure could determine which **`print`** function to call based on which type in a hierarchy of **`shapes`**

- Problems with **`switch`**
  - –Programmer may forget to test all possible cases in a switch.
    - •Tracking this down can be time consuming and error prone
    - •**`virtual`** functions and polymorphic programming can eliminate the need for **`switch`**

Virtual Functions

- **`virtual`** functions
  - Used instead of **`switch`** statements
  - Declaration:
    - Keyword **`virtual`** before function prototype in base class

    `virtual void draw() const;`
  - A base-class pointer to a derived class object will call the correct **`draw`** function
  - If a derived class does not define a **`virtual`** function it is inherited from the base class.

Virtual Functions (cont)

- `ShapePtr->Draw();`
  - –Compiler implements dynamic binding (late binding)
  - –Function determined during execution time

- `ShapeObject.Draw();`
  - –Compiler implements static binding (early binding)
  - –Function determined during compile-time

Abstract and Concrete Classes

- Abstract classes
  - Sole purpose is to provide a base class for other classes
  - No objects of an abstract base class can be instantiated
    - Too generic to define real objects, i.e. **`TwoDimensionalShape`**
    - Can have pointers and references
  - Concrete classes - classes that can instantiate objects
    - Provide specifics to make real objects , i.e. **`Square`**, **`Circle`**

Abstract and Concrete Classes (cont)

- Making abstract classes
  - Declare one or more **`virtual`** functions as "pure" by initializing the function to zero

```
virtual double earnings() const = 0;
```

    - Pure **`virtual`** function

Interface

- An interface is an implementation independent description of the available access functions to an object.
- *Interface classes* in C++ are abstract classes which consist *only of pure virtual functions*

Polymorphism

- Polymorphism:
  – Ability for objects of different classes to respond differently to the same function call
  – Base-class pointer (or reference) calls a **`virtual`** function
    - C++ chooses the correct overridden function in object
  – Suppose **`print`** not a **`virtual`** function

```
Employee e, *ePtr = &e;
HourlyWorker h, *hPtr = &h;
ePtr->print();   //call base-class print function
hPtr->print();   //call derived-class print function
ePtr=&h;         //allowable implicit conversion
ePtr->print();   // still calls base-class print
```

New Classes and Dynamic Binding

- Dynamic binding (late binding)
  - Object's type not needed when compiling **`virtual`** functions
  - Accommodate new classes that have been added after compilation
  - Important for ISV’s (Independent Software Vendors) who do not wish to reveal source code to their customers

Virtual Destructors

- Problem:
  - If base-class pointer to a derived object is **`deleted`**, the base-class destructor will act on the object
- Solution:
  - Declare a **`virtual`** base-class destructor
  - Now, the appropriate destructor will be called

Template

Outline

- Introduction
- Function Templates
- Class Templates
- Class Templates and Non-type Parameters
- Templates and Inheritance
- Templates and friends
- Templates and static Members

Introduction

- Templates
  - Easily create a large range of related functions or classes
  - Function template - the blueprint of the related functions
  - Template function - a specific function made from a function template

Function Templates

- Using function overloading:

```
int addTwoNumbers(int, int);
float addTwoNumbers(float, float);
double addTwoNumbers(double, double);
```

- Disadvantages:

Function Templates Syntax

- The function declaration:

```
template <typename T>  //tell the compiler we are using a template

//T represents the variable type: any fundamental type or user-defined type
T  functionName(T parameter1, T parameter2, ...);
```

- The function definition:

```
template <typename T>
T functionName(T  parameter1, T  parameter2, ...)
{
    function statements;
}
```

Class Templates

- Class templates
  - Allow type-specific versions of generic classes
- Format:

```
template <class T>
class ClassName{
   definition
}
```

- To create an object of the class, type

```
ClassName< type > myObject;
```

Example: `Stack< double > doubleStack;`

Class Templates and Non-type Parameters

- ❖ Can use non-type parameters in templates
  - –Default argument
  - –Treated as **`const`**
- ❖ Example:

```
template< class T, int elements >
Stack< double, 100 > mostRecentSalesFigures;
```

  - • Declares object of type `Stack< double, 100>`

  - –This may appear in the class definition:

```
T stackHolder[ elements ]; //array to hold stack
```

  - • Creates array at compile time, rather than dynamic allocation at execution time

Templates and Inheritance

- A class template can be derived from a class-template specialization.
- A class template can be derived from a non-template class.
- A class-template specialization can be derived from a class-template specialization.
- A non-template class can be derived from a class-template specialization.

Templates and Friends

- Friendships allowed between a class template and
  - Global function
  - Member function of another class
  - Entire class
- **`friend`** functions
  - Inside definition of class template **`X`**:
  - **`friend void f1();`**
    - **`f1()`** a **`friend`** of all template classes
  - **`friend void f2( X< T > & );`**
    - **`f2( X< int > & )`** is a **`friend`** of **`X< int >`** only. The same applies for **`float`**, **`double`**, etc.
  - **`friend void A::f3();`**
    - Member function **`f3`** of class **`A`** is a **`friend`** of all template classes

Templates and Friends (cont)

- **`friend void C< T >::f4( X< T > & );`**
  - **`C<float>::f4( X< float> & )`** is a **`friend`** of **`class X<float>`** only

- **`friend`** classes
  - **`friend class Y;`**
    - Every member function of **`Y`** is a friend with every template class made from **`X`**
  - **`friend class Z<T>;`**
    - Class **`Z<float>`** a **`friend`** of class **`X<float>`**, etc.

Templates and static Members

- Non-template class
  - **`static`** data members shared between all objects
  - **`static`** data members initialized at global scope
- Template classes
  - Each class (**`int`**, **`float`**, etc.) has its own copy of **`static`** data members
  - **`static`** data members initialized at global scope
  - Each template class gets its own copy of **`static`** member functions