TDDE18 & 726G77

Inheritance and polymorphism

Introduction to inheritance

- Inheritance allows us to write functionality once instead of multiple times for multiple classes.
- We can reference a group of classes

```
class Rectangle {
public:
    Rectangle(double h, double w)
         : height{h}, width{w} {}
    double area() const {
        return height * width;
    }
    double get_height() const {
        return height;
    }
    double get_width() const {
        return width;
    }
private:
    double height;
    double width;
};
```

```
class Triangle {
public:
    Triangle(double h, double w)
         : height{h}, width{w} {}
    double area() const {
        return height * width / 2.0;
    }
    double get_height() const {
        return height;
    }
    double get_width() const {
        return width;
    }
private:
    double height;
    double width;
};
```

Introduction to inheritance

- The two classes are totally different, but have a lot of functionality in common.
- Replicate code should be avoided – they increase the risk for bugs
- C++ (and other object oriented languages) have support for creating a common base class that other classes can share.

```
class Shape {
public:
    Shape(double h, double w)
        : height{h}, width{w} {}
    double get_height() const {
          return height;
     }
     double get_width() const {
          return width;
     }
private:
    double width;
    double height;
};
```

Inheritance syntax

The following syntax is used to create a subclass:

```
class <sub-class> : public <base-class> {
...
};
```

```
class Rectangle : public Shape {
public:
    Rectangle(double h, double w)
         : Shape{h, w} {}
    double area() const {
        return height * width;
    }
};
```

```
class Triangle : public Shape {
public:
    Triangle(double h, double w)
         : Shape{h, w} {}
    double area() const {
        return height * width / 2.0;
    }
};
```

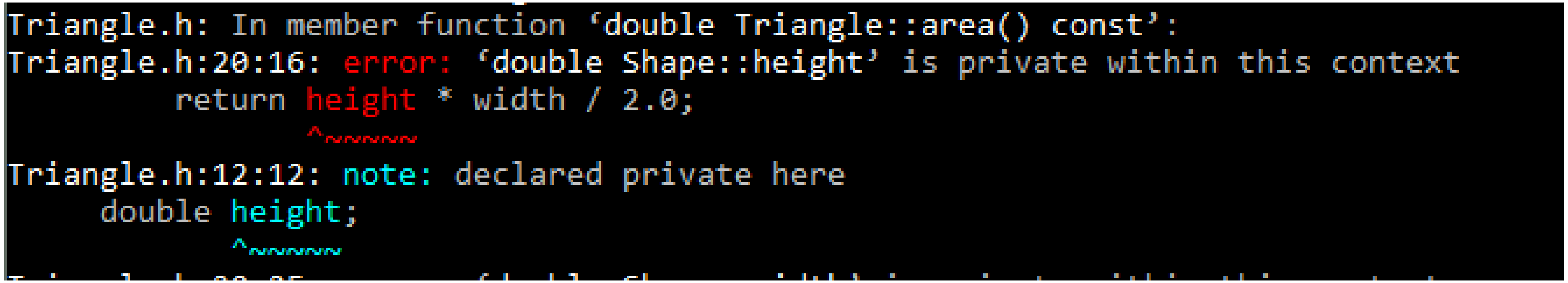
Inheritance

- Inheritance allows us to use a previous class as a model for a new class. All functionality in the original class will be kept (without additional code), and we are allowed to add new functionality.
- The class we use as a model is called the “base class” and the new class we create from this is called “derived class” or “subclass”.
- Inheritance can be done in many levels. One class may be derived from some class, and at the same time base class to another class.

```
class Shape {
public:
    Shape(double h, double w)
        : height{h}, width{w} {}
    double get_height() const {
          return height;
    }
    double get_width() const {
        return width;
    }
private:
    double height;
    double width;
};
```

```
class Triangle : public Shape {
public:
    Triangle(double h, double w)
         : Shape{h, w} {}
    double area() const {
        return height * width / 2.0;
    }
};
```

Compile error – wrong access modifier



```
Triangle.h: In member function ‘double Triangle::area() const’:
Triangle.h:20:16: error: ‘double Shape::height’ is private within this context
        return height * width / 2.0;
               ^~~~~~
Triangle.h:12:12: note: declared private here
     double height;
            ^~~~~~
```

```
class Shape {
public:
    Shape(double h, double w)
        : height{h}, width{w} {}
    double get_height() const {
          return height;
     }
     double get_width() const {
          return width;
     }
private:
    double width;
    double height;
};
```

```
class Triangle : public Shape {
public:
    Triangle(double h, double w)
         : Shape{h, w} {}
    double area() const {
        return get_height() * get_width() / 2.0;
    }
};
```

Class access modifiers

- Public – A public member is accessible from anywhere outside of the class.
- Private – A private member variable or function cannot be accessed, or even viewed from outside the class.
- Protected – A protected member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in derived classes.

```
class Shape {
public:
    Shape(double h, double w)
        : height{h}, width{w} {}
    double get_height() const {
          return height;
    }
    double get_width() const {
        return width;
    }
protected:
    double height;
    double width;
};
```

```
class Triangle : public Shape {
public:
    Triangle(double h, double w)
         : Shape{h, w} {}
    double area() const {
        return height * width / 2.0;
    }
};
```

Public inheritance

This rules apply for the normal public inheritance:

- private members of the base class will neither be accessible in the sub class nor to anyone else
- protected members in the base class become protected also in the subclass, and behave as private to anyone else
- public members in the base class will be public in the sub class

```cpp
class Shape {
public:
    Shape(double h, double w)
        : height{h}, width{w} {}
    double get_height() const {
          return height;
    }
    double get_width() const {
        return width;
    }
protected:
    double height;
    double width;
};
```

```cpp
class Triangle : public Shape {
public:
    Triangle(double h, double w)
          : Shape{h, w} {}
    double area() const {
        return height * width / 2.0;
    }
    // Everything public in Shape
protected:
    // Everything protected in Shape
};
```

Private inheritance

This rules apply for the private inheritance:

- private members of the base class will neither be accessible in the sub class nor to anyone else
- protected members in the base class become private in the subclass, and behave as private to anyone else
- public members in the base class will be private in the sub class and behave as private to anyone else

```
class Shape {
public:
    Shape(double h, double w)
        : height{h}, width{w} {}
    double get_height() const {
          return height;
    }
    double get_width() const {
        return width;
    }
protected:
    double height;
    double width;
};
```

```
class Triangle : private Shape {
public:
    Triangle(double h, double w)
         : Shape{h, w} {}
    double area() const {
        return height * width / 2.0;
    }
private:
    // Everything public and protected in Shape
};
```

Protected inheritance

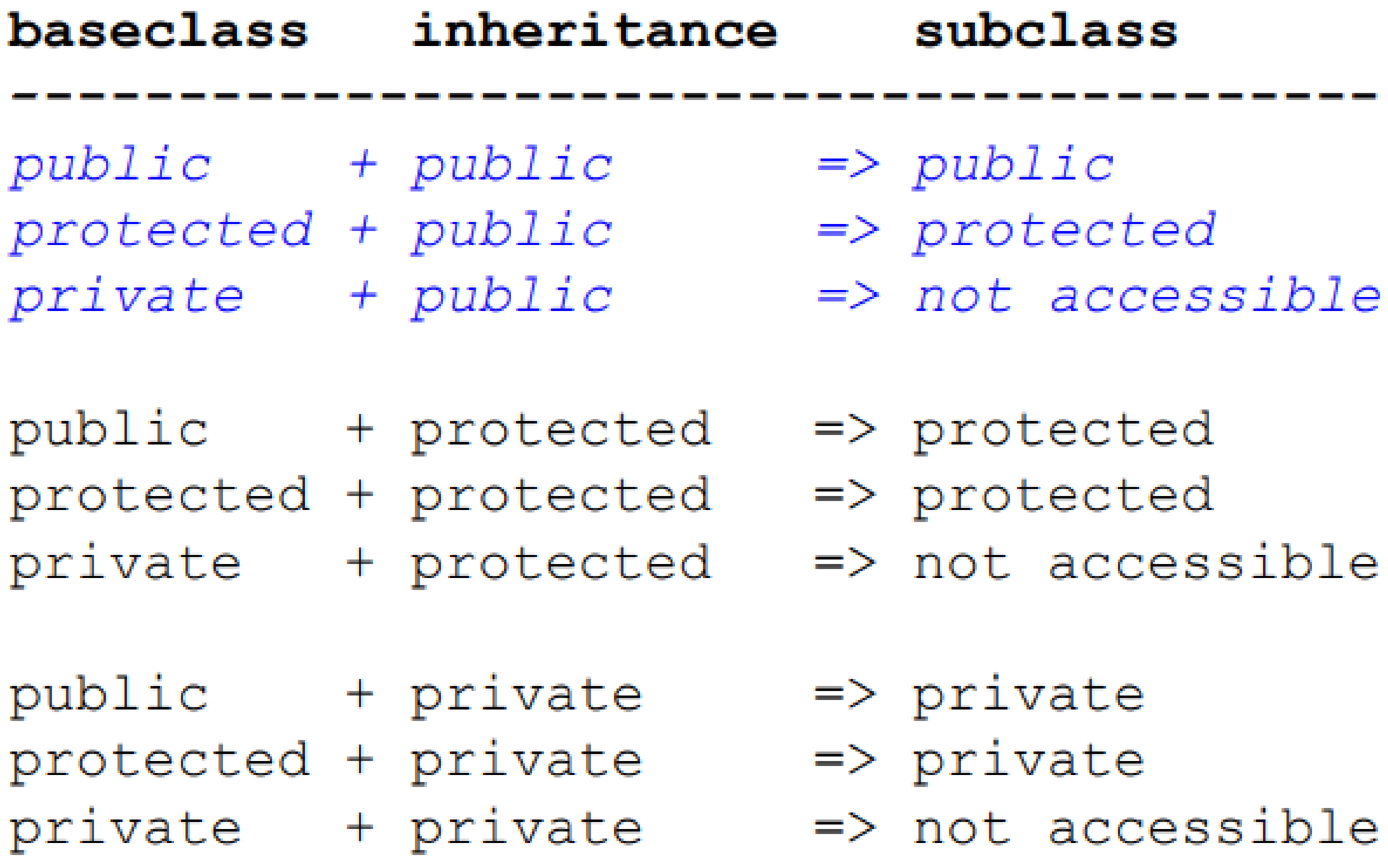
This rules apply for the protected inheritance:

- private members of the base class will neither be accessible in the sub class nor to anyone else
- protected members in the base class become protected in the subclass, and behave as private to anyone else
- public members in the base class will become protected in the sub class and behave as private to anyone else

```
class Shape {
public:
    Shape(double h, double w)
        : height{h}, width{w} {}
    double get_height() const {
          return height;
    }
    double get_width() const {
        return width;
    }
protected:
    double height;
    double width;
};
```

```
class Triangle : protected Shape {
public:
    Triangle(double h, double w)
         : Shape{h, w} {}
    double area() const {
        return height * width / 2.0;
    }
protected:
    // Everything public and protected in Shape
};
```

Inheritance table

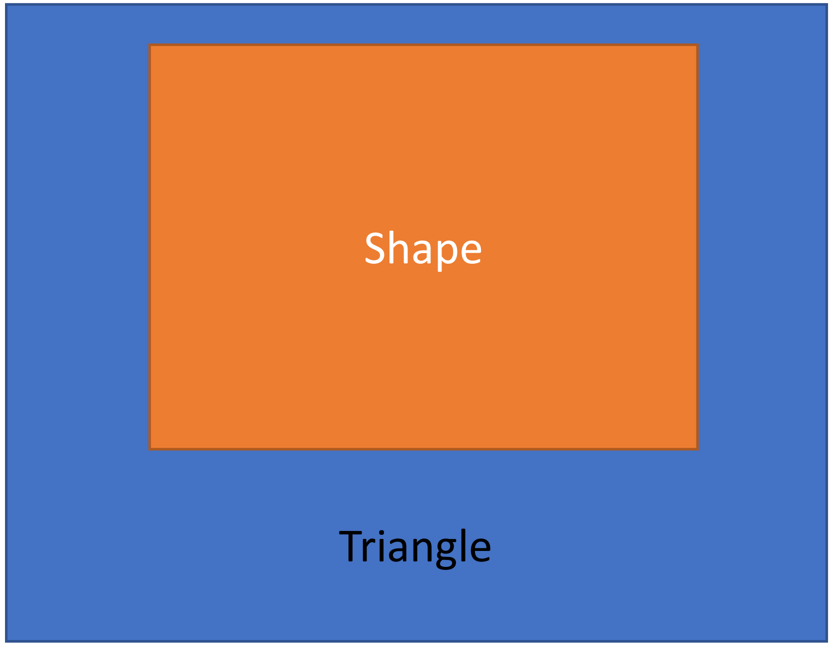


baseclass		**inheritance**		**subclass**
public	*+*	*public*	*=>*	*public*
protected	*+*	*public*	*=>*	*protected*
private	*+*	*public*	*=>*	*not accessible*
public	+	protected	=>	protected
protected	+	protected	=>	protected
private	+	protected	=>	not accessible
public	+	private	=>	private
protected	+	private	=>	private
private	+	private	=>	not accessible

We will only use public inheritance in the course, outlined in italic

Initialization of derived classes

- When creating an object of an derived, the inner part (base class) must be initialized first.
- It is common for the constructor of the derived class to call the constructor of the base class.



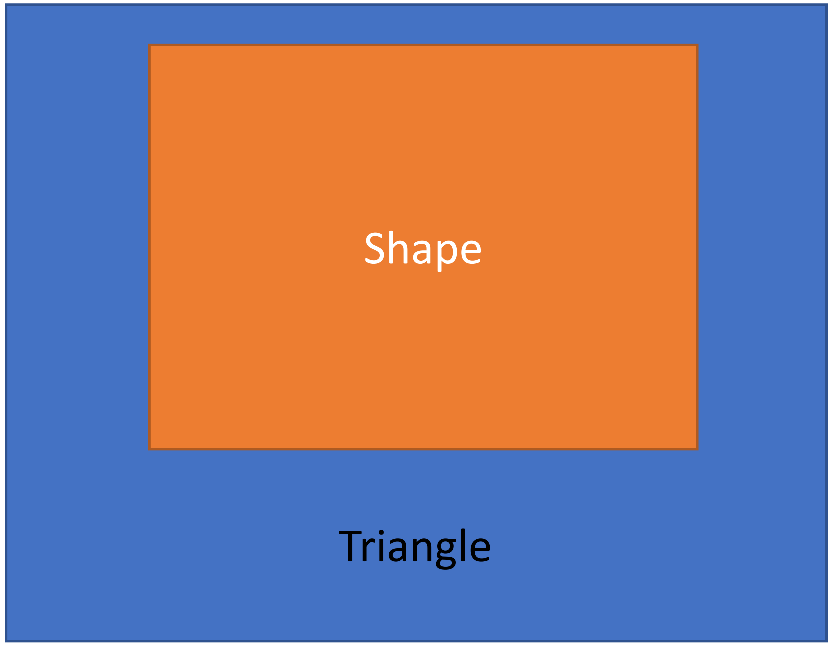
Calling base constructor

This must be done with an initialization list

```
<sub-class>::<sub-class>(<param-list>)
    : <base-class>(<argument-list>),
    <member-name>(<argument>)
{
    <constructor-code>
}
```

Initialization of derived classes

```
class Triangle : public Shape {
public:
    Triangle(double h, double w)
         : Shape{h, w} {}
    ...
};
```



How to use a derived class

- Given the public member functions from both classes:

```
int main() {
    Triangle t{12, 4};
    cout << t.get_height() << “ “ << t.area() << endl;
}
```

Function arguments

```
void foo(Triangle const& t) {
    cout << t.get_height() << endl;
}

void foo(Rectangle const& r) {
    cout << r.get_height() << endl;
}

int main() {
   Triangle t{12, 4};
   foo(t);
   Rectangle r{24, 8};
   foo(r);
}
```

Function arguments

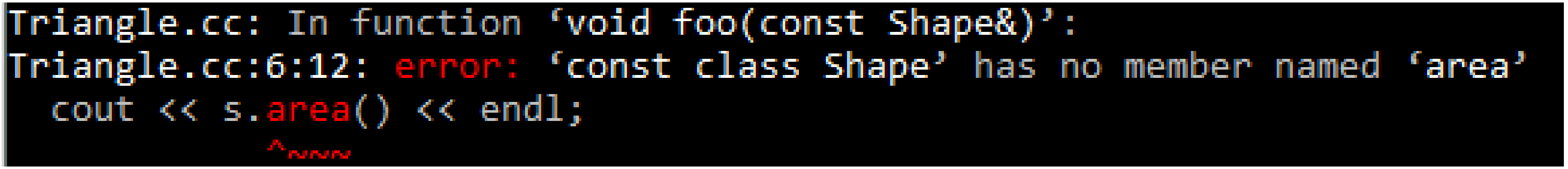
If we create a function that takes a reference to Shape then we can send both Triangle and Rectangle. This gives us less duplicate code!

```
void foo(Shape const& s) {
    cout << s.get_height() << endl;
}

int main() {
   Triangle t{12, 4};
   foo(t);
   Rectangle r{24, 8};
   foo(r);
}
```

What about the function *area*?

```
void foo(Shape const& s) {
    cout << s.area() << endl;
}
```



```
Triangle.cc: In function ‘void foo(const Shape&)’:
Triangle.cc:6:12: error: ‘const class Shape’ has no member named ‘area’
  cout << s.area() << endl;
            ^~~~
```

```
class Shape {
public:
    ...
    double area() const {
        return 0;
    }
    ...
};
```

```
class Triangle : public Shape {
public:
    Triangle(double h, double w)
         : Shape{h, w} {}
    double area() const {
        return height * width / 2.0;
    }
};
```

What about the function *area*?

```
void foo(Shape const& s) {
    cout << s.area() << endl;
}

int main() {
   Triangle t{12, 4};
   foo(t);            // print out 0
}
```

Polymorphism

- When we in addition to inheritance use polymorphism (poly = many, morph = shifting) we can modify or customize the behavior of the base class. Thus we can have one class with behavior that differ depending on which subclass it actually is.
- The exact behavior is not determined when compiling the program, but when the program runs (at runtime).
- To enable polymorphism the base class must declare the morphing member functions as virtual.

Polymorphism

- With the keyword virtual we can declare in the base class a member that the subclasses can override

```
class Shape {
public:
    ...
    virtual double area() const {
        return 0;
    }
    ...
};
```

What about the function *area*?

```
void foo(Shape const& s) {
    cout << s.area() << endl;
}

int main() {
   Triangle t{12, 4};
   foo(t);                  // print out 24
}
```

Enabling polymorphism

- C++ doesn’t use polymorphism as a default. The programmer must opt-in for this feature.
- Use the keyword virtual for the member function that you want to allow polymorphism.
- You must use either a pointer to the base class or a reference to the base class.

Enabling polymorphism

```
int main() {
   Triangle t{12, 4};
   t.area();              // 24
   Shape s1{t};
   s1.area();             // 0
   Shape & s2{t};
   s2.area()              // 24
   Shape * s3{&t};
   s3->area();            // 24
}
```

Polymorphism – how does it work

- You usually talk about two different types – static types and dynamic types.

```
Triangle t{12, 4};
Shape & s{t};
```

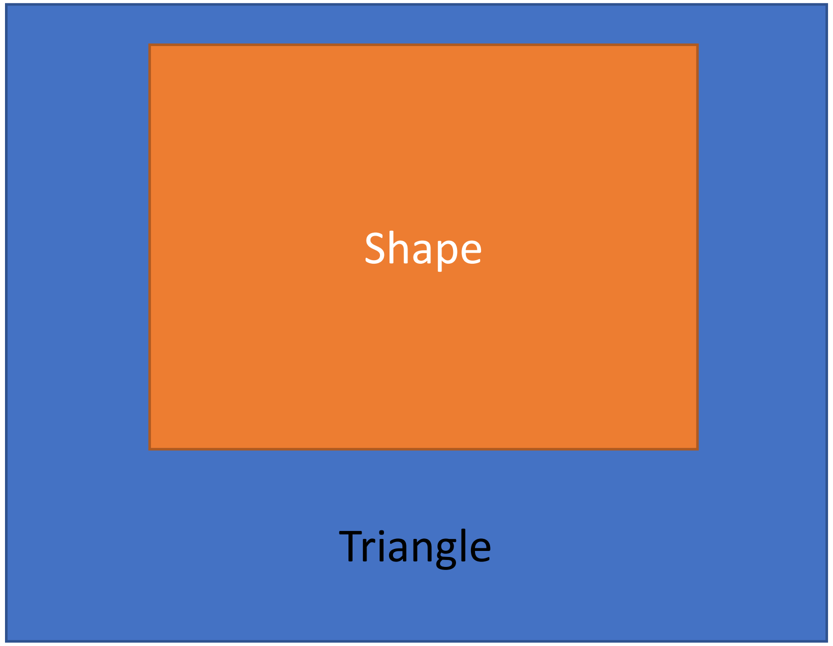
- The static type of s is always `Shape &`
- The dynamic type depends on what s is referring to, in this case `Triangle`

Polymorphism – how does it work

- When calling a member function, the compiler does the following:
    - If the static type isn’t of pointer type or reference type => Call the function in the static type.
    - If the function is not virtual => Call the function in the static type.
    - Otherwise => Call the function in the dynamic type.

Destruction of derived classes

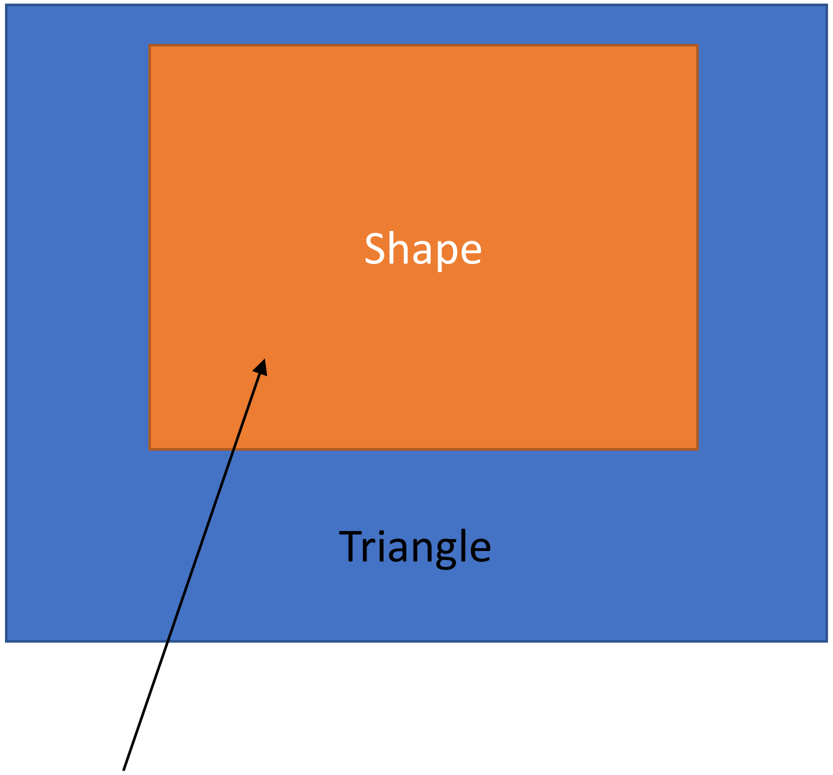
- When destroying an object of an derived, the outer part (subclass) must be destroyed first.
- It is a must for the destructor of the base class to be virtual.



Destruction of derived classes

```
class Shape {
   ...
   ~Shape() {}
   ...
};

int main() {
    Shape * s{new Triangle{4, 2}};
    delete s;
}
```

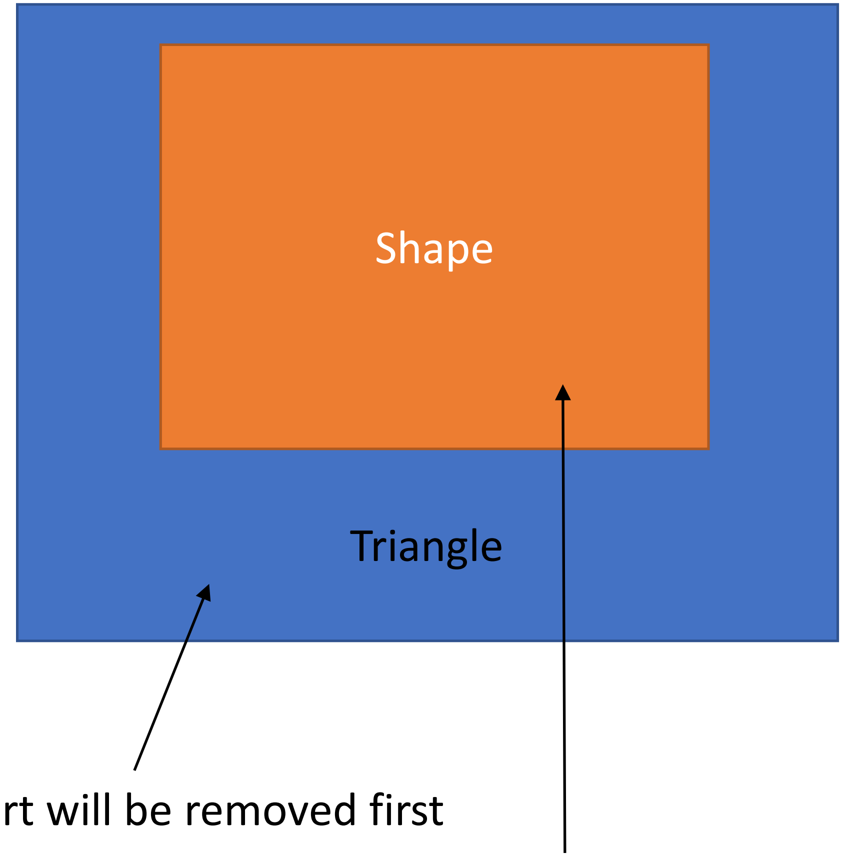


Only this part will be removed

Destruction of derived classes

```
class Shape {
   ...
   virtual ~Shape() {}
   ...
};

int main() {
    Shape * s{new Triangle{4, 2}};
    delete s;
}
```



This part will be removed first

This part will be removed second

Pure virtual & Abstract class

```
    virtual double area() const {
        return 0;
    }

// change it to


    virtual double area() = 0;
```

- This implementation makes no sense.
- But if this function is missing we get a compile error.
- Fix is to make this a pure virtual function and the class an abstract class

Pure virtual & Abstract class

- Abstract classes are used to represent general concepts (for example, Shape), which can be used as base classes for concrete classes (for example, Triangle).
- No objects of an abstract class can be created. Abstract types cannot be used as parameter types, as function return types, or as the type of an explicit conversion.
- Pointers and references to an abstract class can be declared.

Pure virtual & Abstract class

```
class Shape {
public:
    ...
    double area() const = 0;
    ...
};
```

```
int main() {
    Shape s;        // Error: abstract class
    Triangle t{12, 4}; // OK
    Shape s2{t};    // Error abstract class.
    Shape & s2{t}; // OK to reference abstract class
    Shape * s3{&t}; // Ok to point to abstract class
}
```

Pure virtual & Abstract class

```
class Shape {
public:
    ...
    int corners() const = 0;
    ...
};
```

- Subclasses must implement the pure virtual functions or they will become abstract classes too.

```
int main() {
   Triangle t{12, 4};
   // Error: Abstract class. Missing corner function
}
```

Keyword *Override*

```
class Shape {
public:
    ...
    virtual double area() const {
        return 0;
    }
    ...
};
```

```
class Triangle: public Shape {
public:
    Triangle(double radius, double w)
         : Shape{h, w} {}
    double ara() const {
        return height * width / 2.0;
    }
};
```

```
int main() {
    Triangle t{12, 4};
    Shape & s{t};
    s.area();                    // 0
}
```

Keyword *Override*

```
class Shape {
public:
    ...
    virtual double area() const {
        return 0;
    }
    ...
};
```

```
class Triangle: public Shape {
public:
    Triangle(double radius, double w)
         : Shape{h, w} {}
    double ara() const {
        return height * width / 2.0;
    }
};
```

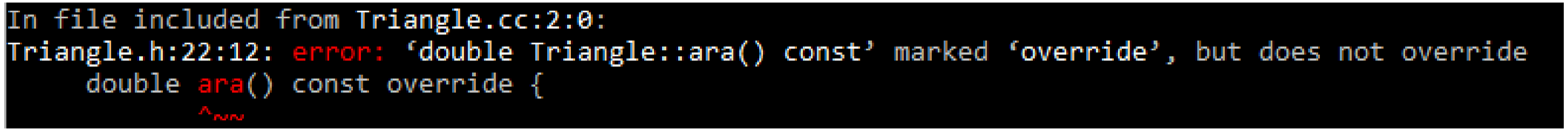
```
int main() {
    Triangle t{12, 4};
    Shape & s{t};
    s.area();                        // 0
}
```

Typo

Keyword *Override*

```
class Shape {
public:
    ...
    virtual double area() const {
        return 0;
    }
    ...
};
```

```
class Triangle: public Shape {
public:
    Triangle(double radius, double w)
         : Shape{h, w} {}
    double ara() const override {
        return height * width / 2.0;
    }
};
```



```
In file included from Triangle.cc:2:0:
Triangle.h:22:12: error: ‘double Triangle::ara() const’ marked ‘override’, but does not override
     double ara() const override {
            ^~~
```

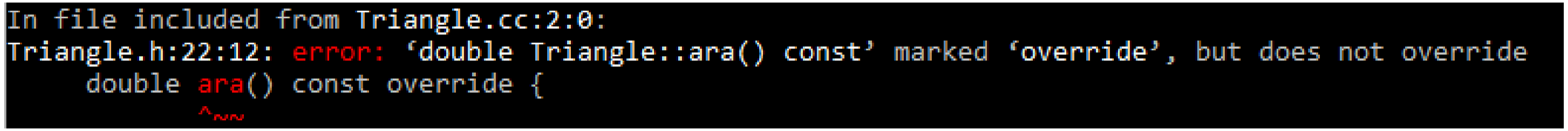
Keyword *Override*

- In a member function declaration or definition, *override* ensures that the function is virtual and is overriding a virtual function from a base class. The program is ill-formed (a compile-time error is generated if this is not true.

Keyword *Override*

```
class Shape {
public:
    ...
    virtual double area() const {
        return 0;
    }
    ...
};
```

```
class Triangle: public Shape {
public:
    Triangle(double radius, double w)
          : Shape{h, w} {}
    double ara() const override {
        return height * width / 2.0;
    }
};
```

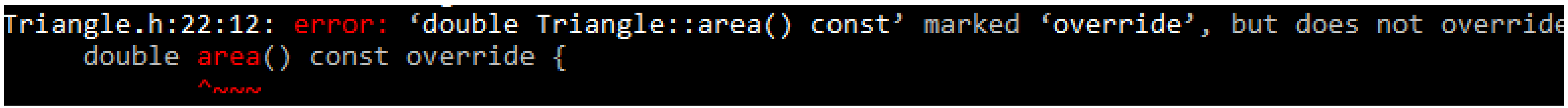


```
In file included from Triangle.cc:2:0:
Triangle.h:22:12: error: ‘double Triangle::ara() const’ marked ‘override’, but does not override
     double ara() const override {
            ^~~
```

Keyword *Override*

```
class Shape {
public:
    ...
    double area() const {
        return 0;
    }
    ...
};
```

```
class Triangle: public Shape {
public:
    Triangle(double radius, double w)
         : Shape{h, w} {}
    double area() const override {
        return height * width / 2.0;
    }
};
```



```
Triangle.h:22:12: error: ‘double Triangle::area() const’ marked ‘override’, but does not override
    double area() const override {
           ^~~~
```

Using declaration

- Using-declarations can be used to introduce members into other block scopes, or to introduce base class members into derived class definitions.

```
using namespace std;
using std::cin;
```

Using declaration in class definition

- Using-declaration introduces a member of a base class into the derived class definition, such as to expose a protected member of base as public member of derived.

```
class Shape {
public:
    Shape(double h, double w)
        : height{h}, width{w} {}
    double get_height() const {
          return height;
    }
    double get_width() const {
        return width;
    }
protected:
    double height;
    double width;
};
```

```
class Rectangle : public Shape {
public:
    Rectangle(double h, double w)
         : Shape{h, w} {}
    double area() const {
        return height * width;
    }
    using Shape::height;
};
```

height is now public

```
class Shape {
public:
    Shape(double h, double w)
        : height{h}, width{w} {}
    double get_height() const {
          return height;
    }
    double get_width() const {
        return width;
    }
protected:
    double height;
    double width;
};
```

```
class Rectangle : public Shape {
public:
    Rectangle(double h, double w)
         : Shape{h, w} {}
    double area() const {
        return height * width;
    }
    using Shape::height;
};
```

```
class Square : public Rectangle {
...
private:
    using Shape::height;
}
```

Using declaration for constructors

- The derived class can copy in all the constructors from the base class with a using-declaration and use it as its own.

```
class Rectangle : public Shape {
public:
    using Shape::Shape;
    double area() const {
        return height * width;
    }
    using Shape::height;
};
```

It is possible to create a Rectangle object with height and width as input arguments.

`Rectangle r{12, 3};`

dynamic_cast

- dynamic_cast can only be used with pointers and references to classes. Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type.
- This naturally includes pointer upcast (converting from pointer-to-derived to pointer-to-base), in the same way as allowed as an implicit conversion.
- dynamic_cast can also downcast (convert from pointer_to_base to pointer_to_derived) polymorphic classes (those with virtual members).

downcasting

- Often you would like to downcast whenever you want to get a specific specialized functionality in a derived class.

```
Triangle t{12, 3};
Shape * s{t};
s->area_formula(); // Error
Triangle * t_ptr{dynamic_cast<Triangle*>(s)};
t_ptr->area_formula(); // Ok
```

```
class Triangle: public Shape {
public:
    Triangle(double radius, double w)
         : Shape{h, w} {}
    string area_formula() const {
        return “height * width / 2.0”;
    }
};
```

downcasting – wrong type

- dynamic_cast will return nullptr if it cannot downcast to that type

```
Triangle t{12, 3};
Shape * s{t};
s->area_formula(); // Error
Rectangle * r_ptr{dynamic_cast<Rectangle*>(s)};
if (r_ptr != nullptr) {
   r_ptr->area_formula(); // Will never go here
}
```

Type alias

- A type alias declaration introduces a name which can be used as a synonym for the type denoted. It does not introduce a new type and it cannot change the meaning of an existing type name.
- The type alias will behave exactly as the type denoted.

```
using FirstName = string;
FirstName f1{“Sam”};
f1.size(); // returns 3
```