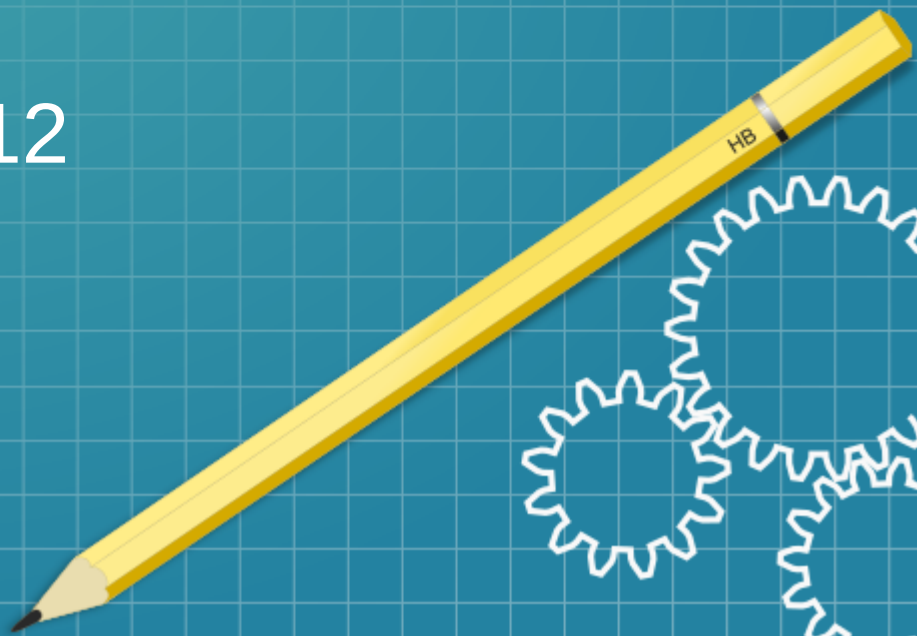
Object-Oriented Programming: Polymorphism

Chapter 12



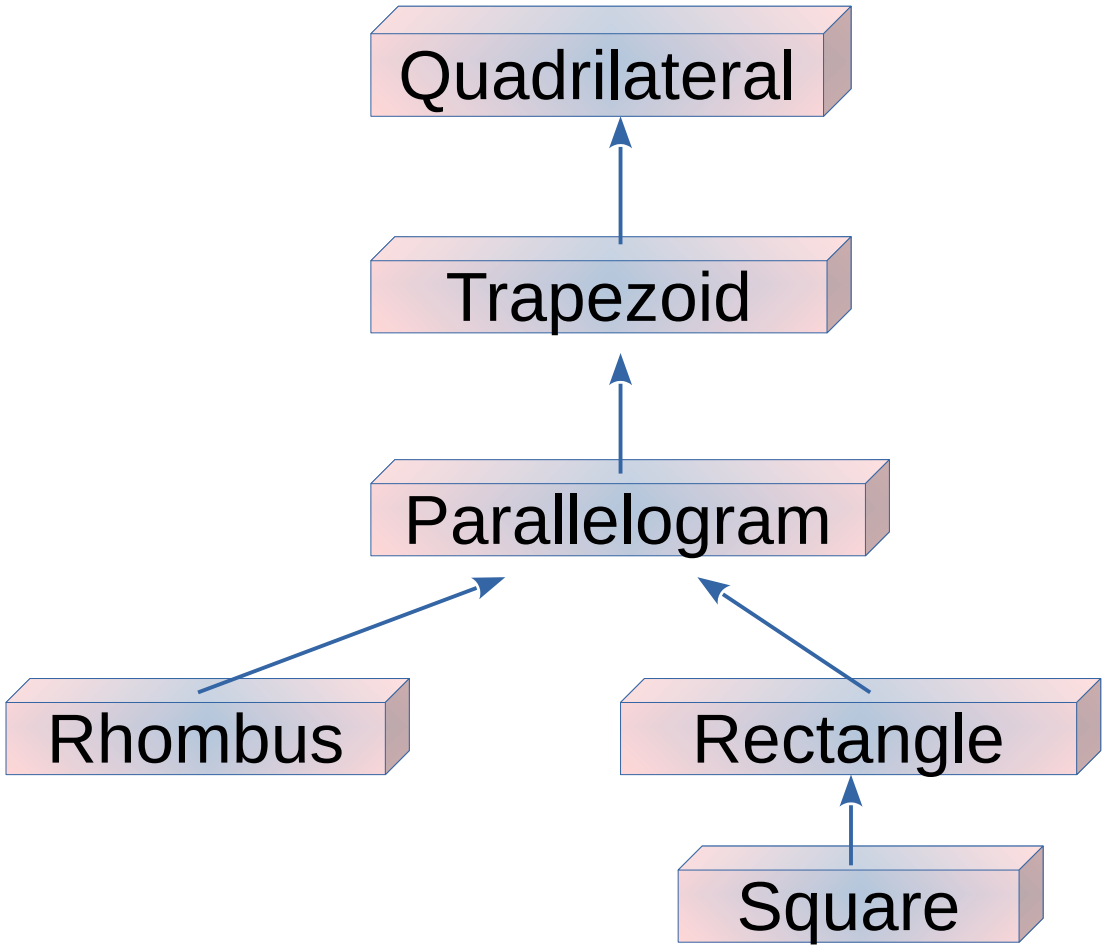
Polymorphism : Introduction

Polymorphism allows the programmer to treat derived class members just like their parent class's members.

With polymorphism we can design and implement *easily extensible systems*, i.e. new classes can be added with little or no modification to the general portions of the program, as longs as the new classes are part of the inheritance hierarchy that the program processes generally.

Polymorphism : Introduction

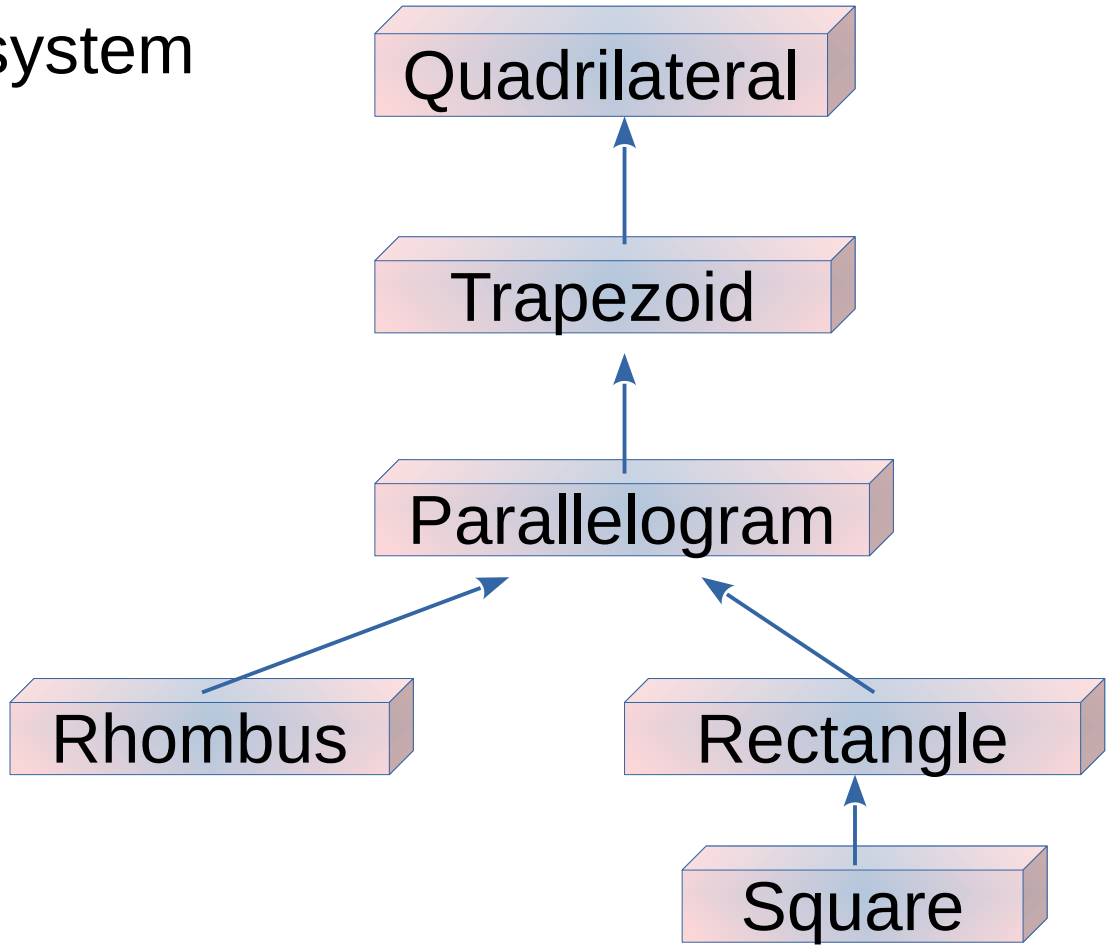
Recall our Quadrilateral Hierarchy:



Polymorphism : Introduction

Recall our Quadrilateral Hierarchy:

I’m planning to design a system that will be drawing these geometric figures.

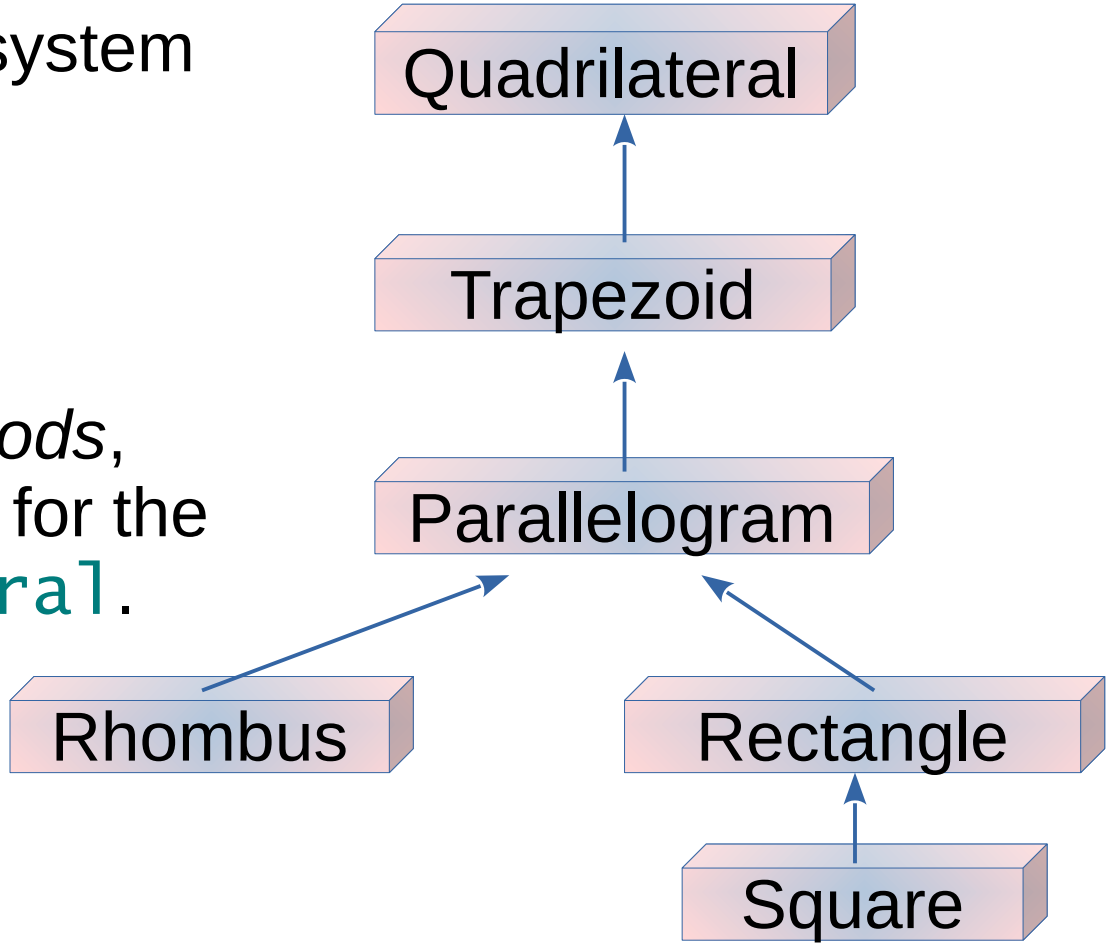


Polymorphism : Introduction

Recall our Quadrilateral Hierarchy:

I'm planning to design a system that will be drawing these geometric figures.

So I will introduce more *attributes* and more *methods*, including a method `draw` for the base class `Quadrilateral`.



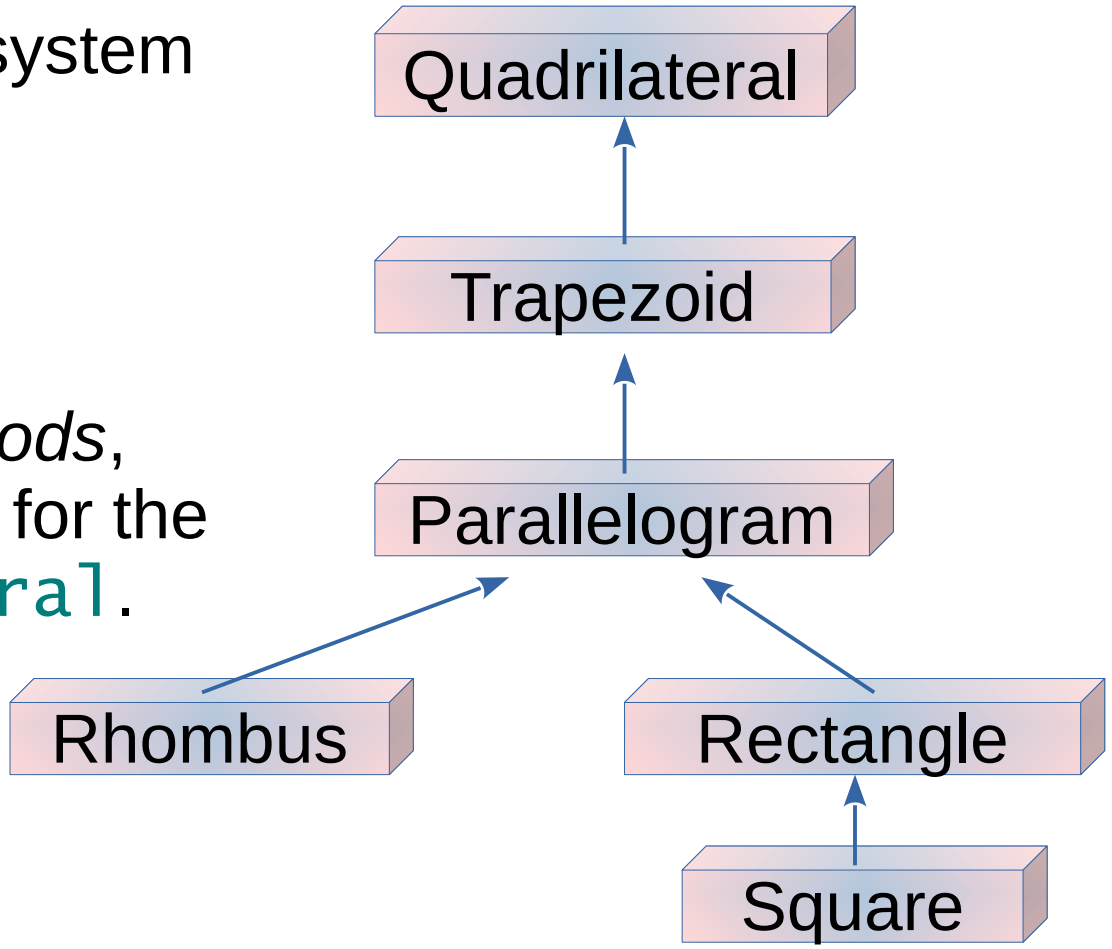
Polymorphism : Introduction

Recall our Quadrilateral Hierarchy:

I'm planning to design a system that will be drawing these geometric figures.

So I will introduce more *attributes* and more *methods*, including a method `draw` for the base class `Quadrilateral`.

For each of the derived classes, `Trapezoid`, `Parallelogram`, etc



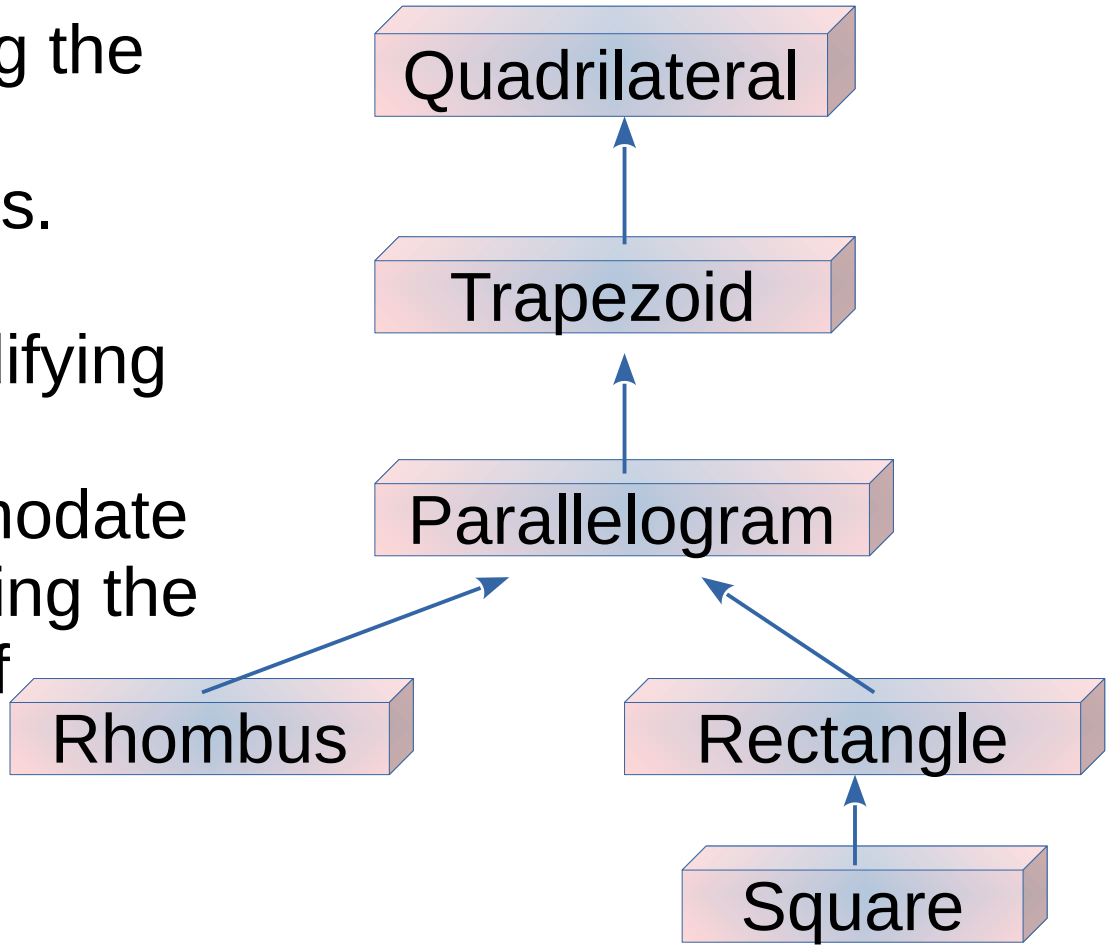
I will need to implement the function `draw`. So each object will respond differently to the same message: `draw`.

Polymorphism : Introduction

Recall our Quadrilateral Hierarchy:

My system will be sending the same message: `draw` to elements of different types.

In the future, without modifying the system, I can use polymorphism to accommodate additional classes, including the ones I didn't even think of at this moment!



Relationships between objects in an Inheritance Hierarchy

Let's consider a series of examples to see how *base/parent class* and *derived/child class pointers* can be aimed at objects and how can they be manipulated.

- see Example1.cpp and the conclusion in the comment at the end
- see Example2.cpp and the conclusion in the comment at the end
- see Example3.cpp and the conclusion in the comment at the end

Relationships between objects in an Inheritance Hierarchy

Downcasting:

We saw in the examples 1-3 that *base/parent class pointer* can be “aiming” at a *derived/child class object*, but it can invoke only functions defined in the *base/parent class*.

We can cast (downcast) such a pointer to a derived class pointer, however, it could be dangerous.

to be discussed

Virtual Functions and Virtual Destructors

Consider a situation when our driver-program draws different kinds of shapes: quadrilaterals, trapezoids, rhombuses, squares, …

It would be useful to treat all the shapes *generally*, as objects of base class `Quadrilateral`, and use the base-class `Quadrilateral` pointer to invoke the function `draw`, and allow the program to determine *dynamically* (at runtime) which derived-class `draw` function to use (based on the type of the object to which the pointer points at this particular moment)!

This is a *polymorphic behavior*!

With *virtual functions*, the type of the object, not the type of the handle used to invoke the object's member function, determines which version of a virtual function to invoke.

Virtual Functions and Virtual Destructors

With *virtual functions*, the type of the object, not the type of the handle used to invoke the object's member function, determines which version of a virtual function to invoke.

Syntax example:

```
virtual void draw() const;
```

Virtual Functions and Virtual Destructors

With *virtual functions*, the type of the object, not the type of the handle used to invoke the object's member function, determines which version of a virtual function to invoke.

Syntax example:

```
virtual void draw() const override;
```

We can add the keyword `override` (starting from C++ 11) to the prototype of every *child/derived-class function* that overrides a *base-class virtual function*.

- this will ensure that we override a *base/parent-class function* with the appropriate signature, and
- this will prevent us from hiding a *base/parent-class function* that has the same name and different signature

see virtualFunctionsExample.h and virtualFunctionsExample.cpp

Virtual Functions and Virtual Destructors

A similar situation is with destructors, when working with dynamically allocated memory:
if a *derived-class object* with a *non-virtual destructor* is destroyed by applying the `delete` operator to a *base-class pointer to the object*, the C++ standard specifies that the *behavior is undefined*.

Hence, announce the base-class destructor as virtual:

```
virtual ~Quadrilateral(…) {};
```

When a *derived-class object* is destroyed, both destructors (the derived and base class's) execute.

Virtual Functions and Virtual Destructors

A similar situation is with destructors, when working with dynamically allocated memory:
if a *derived-class object* with a *non-virtual destructor* is destroyed by applying the `delete` operator to a *base-class pointer to the object*, the C++ standard specifies that the *behavior is undefined*.

Hence, announce the base-class destructor as virtual:

```
virtual ~Quadrilateral(…) {};
```

When a *derived-class object* is destroyed, both destructors (the derived and base class's) execute.

Note that constructors cannot be virtual !

`final` member functions and classes

Starting from C++ 11, if we want to announce that a function should not be overridden in child classes,we announce it as `final` in its prototype:

```
virtual play(…) final;
```

So it will be used by all objects of the parent class and all objects of child classes.

`final` member functions and classes

Starting from C++ 11, if we want to announce that a function should not be overridden in child classes,we announce it as `final` in its prototype:

```
virtual play(…) final;
```

So it will be used by all objects of the parent class and all objects of child classes.

Similarly, if we don't want a class to be used as parent/base class, we will announce it as `final`:

```
class myClass final {
// this class cannot be a base class
…
};
```

HW assignment

2) Explore the idea of making functions of `Area` and `Perimeter` as *virtual functions* in our `Quadrilateral` class hierarchy that so far consists of classes `Quadrilateral`, `Trapezoid`, `Rectangle` and `Square`.

What do you need to do for it?

Do you need to also announce these functions as virtual in `Quadrilateral` and `Trapezoid` classes?

Do it!

Self-Study:
read sections 12.6 – 12.7

Optional (for self-development):
sections 12.8 and 12.9