Classes: inheritance, polymorphism and encapsulation

Plan for today

- We will talk about:
  - Inheritance
  - Polymorphism
  - Hierarchies
  - Data hiding (encapsulation)
  - Has-a vs is-a relationship

Data hiding or encapsulation

- Data should be private:
  - So it will not be changed inadvertently
  - Use `private` data, and pairs of `public` access functions to get and set the data, if needed
- Our functions can be private or public
  - `public` for interface
  - `private` for functions used only internally to a class

What does “private” buy us?

- We can change our implementation after release
- We don’t expose any libraries we used in representation to our users
  - We could replace them with other libraries without affecting user code
- Functional interfaces can be nicer to read and use
  - Example: s.add(x) rather than s.points.push_back(x)
- We can enforce immutability of objects
  - Or allow only some types of changes (only color and style change; not the shape)
  - `const` member functions
- The value of this “encapsulation” varies with application domains
  - Is often most valuable
  - Is the ideal, hide representation unless you have a good reason not to

Access

- C++ provides a simple model of access to members of a class. A member of a class can be:
  - `private`: its name can be used only by members of the class in which it is declared
  - `protected`: its name can be used only by members of the class in which it is declared and members of classes derived from that
  - `public`: its name can be used by all functions

These definitions ignore the concept of “friend” that I introduced to us, as well as a few minor details.

Inheritance

- Derivation is a way to build one class from another so that the new class can be used in place of the original.
- The derived class inherits all of the members of its base class
- Other names of the derived class:
  - subclass (and base class is called “superclass”)
  - child class (and the base class is called parent class or ancestor) – *less formal*

Inheritance

- Derivation is a way to build one class from another so that the new class can be used in place of the original.
- The derived class inherits all of the members of its base class
- Other names of the derived class:
  - subclass (and base class is called "superclass")
  - child class (and the base class is called parent class or ancestor) – *less formal*
- Why use inheritance?
  - It reduces the duplication of existing code, and
  - It can save time during program development by taking advantage of proven, high-quality, already defined classes

Inheritance

- The derived class may
  - introduce one or more behaviors beyond those that are inherited (augmenting the base class)
  - specialize one or more of the inherited behaviors from the base class (provide an alternative definition for the inherited method, i.e. override the original definition)

Inheritance

- The derived class may
  - introduce one or more behaviors beyond those that are inherited (augmenting the base class)
  - specialize one or more of the inherited behaviors from the base class (provide an alternative definition for the inherited method, i.e. override the original definition)
- A single class can serve as base class for many derived classes
- A single derived class can inherit from multiple base classes (multiple inheritance)

Object-oriented programming

- The use of *inheritance*, *run-time polymorphism*, and *encapsulation* is the most common definition of *object-oriented programming*.
- C++ directly supports object-oriented programming
  - In addition to other programming styles
- C++ supports generic programming
  - when classes or functions can be parameterized over a type
  - recall template classes and template functions

When inheriting

- C++ has `private`, `protected`, and `public` inheritance

When inheriting

- C++ has `private`, `protected`, and `public` inheritance

```
class D: access-specifier base-class
```

When inheriting

- C++ has `private`, `protected`, and `public` inheritance

  `class D: access-specifier base-class`

  - If a base of class D is `private`, its `public` and `protected` members can be used only by members of D

When inheriting

- C++ has `private`, `protected`, and `public` inheritance

  `class D: access-specifier base-class`

  - If a base of class D is `private`, its `public` and `protected` members can be used only by members of D
  - If a base of class D is `protected`, its `public` and `protected` member names can be used only by members of D and members of classes derived from D

When inheriting

- C++ has `private`, `protected`, and `public` inheritance

  `class D: access-specifier base-class`

  - If a base of class D is `private`, its `public` and `protected` members can be used only by members of D
  - If a base of class D is `protected`, its `public` and `protected` member names can be used only by members of D and members of classes derived from D
  - If a base is `public`, its `public` member names become public members of D, the protected members of the base class become protected members of D. A base class's private members are never accessible directly from a derived class D.

When inheriting

- C++ has `private`, `protected`, and `public` inheritance

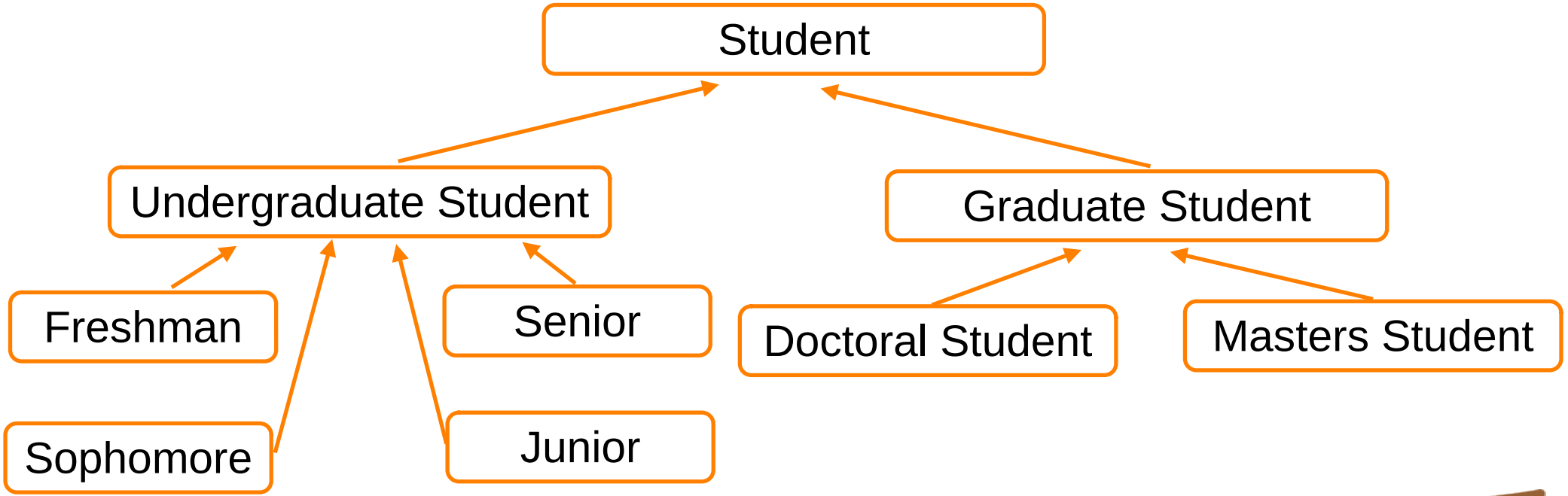
  `class D: access-specifier base-class`

  - If a base of class D is `private`, its `public` and `protected` members can be used only by members of D
  - If a base of class D is `protected`, its `public` and `protected` member names can be used only by members of D and members of classes derived from D
  - If a base is `public`, its `public` member names become public members of D, the protected members of the base class become protected members of D. A base class's private members are never accessible directly from a derived class D.

When access specifier is not used, it is `private` by default

Class Hierarchy

- Inheritance relationships form class hierarchies
- We can look at colleges and universities and build a student inheritance hierarchy:



is-a vs has-a relationships

- The relationship between a base class and derived class is often termed as is-a relationship, meaning that the object of the derived class also can be treated as an object of its base class.

is-a vs has-a relationships

- The relationship between a base class and derived class is often termed as is-a relationship, meaning that the object of the derived class also can be treated as an object of its base class.
  - square (derived) is a quadrilateral (base)
  - Junior (derived) is an Undergraduate Student (base)

is-a vs has-a relationships

- The relationship between a base class and derived class is often termed as is-a relationship, meaning that the object of the derived class also can be treated as an object of its base class.
  - square (derived) is a quadrilateral (base)
  - Junior (derived) is an Undergraduate Student (base)
- When a class is implemented using an instance variable of another, it is termed as has-a relationship.

is-a vs has-a relationships

- The relationship between a base class and derived class is often termed as is-a relationship, meaning that the object of the derived class also can be treated as an object of its base class.
  - square (derived) is a quadrilateral (base)
  - Junior (derived) is an undergraduate student (base)
- When a class is implemented using an instance variable of another, it is termed as has-a relationship.
  - `class MixedNumber` can have objects of types `int` and `Rational` as its attributes
  - `class Rational` can have objects of type int as numerator and denominator

is-a vs has-a relationships

- there is not always a clear-cut rule for when to use *inheritance* and when to use *has-a relationship*.

- the decision comes down to the number of potentially *inherited behaviors* that are undesirable versus the number of desirable ones that would need to be explicitly regenerated if using a *has-a relationship*.

Example: a bank account

- Let’s think about a design of a very basic bank account class:

Example: a bank account

- Let’s think about a design of a very basic bank account class:
  - there should be a person’s record on file
  - A balance
  - A due date

Example: a bank account

- Let's think about a design of a very basic bank account class:
  - there should be a person's record on file
  - A balance
  - A due date

```
struct PersonInfo
{
   string fullName;
   Date birthday;
};
```

Example: a bank account

- Let's think about a design of a very basic bank account class:
  - there should be a person's record on file
  - A balance
  - A due date

```
struct PersonInfo
{
   string fullName;
   Date birthday;
};
```

Date.h

Example: a bank account

- Let's think about a design of a very basic bank account class:
  - there should be a person's record on file
  - A balance
  - A due date

```
struct PersonInfo
{
   string fullName;
   Date birthday;
};
```

Date.h

```
class Account
{
   Date dueDate;
   PersonInfo p;
   double balance;

public:
   Account(...);
   double getBalance();
   void deposit(double a);
   bool withdraw(double a);
};
```

Example: a bank account

- Let's think about a design of a very basic bank account class:
  - there should be a person's record on file
  - A balance
  - A due date

```
struct PersonInfo
{
   string fullName;
   Date birthday;
};
```

Date.h

```
class Account
{
   Date dueDate;
   PersonInfo p;
   double balance;

public:
   Account(...);
   double getBalance();
   void deposit(double a);
   bool withdraw(double a);
};
```

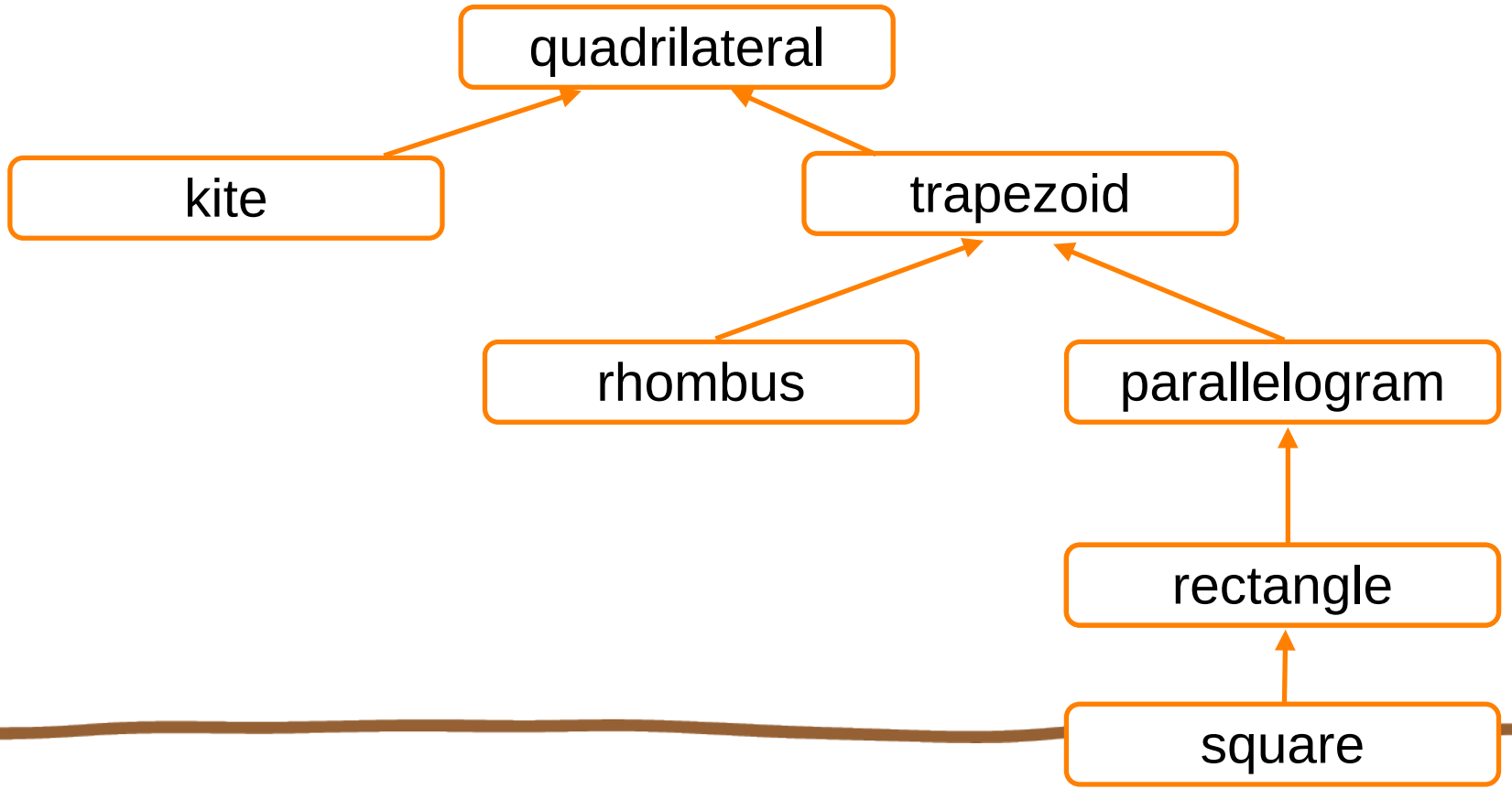
Account class is using an instance variables of PersonInfo struct, and class Date – has-a relationship

Polymorphism

- Let’s think about quadrilaterals: polygons having four sides, four angles, and four vertices.
- Quadrilaterals can be classified into parallelograms, squares, rectangles, and rhombuses, trapezoids, and kites.

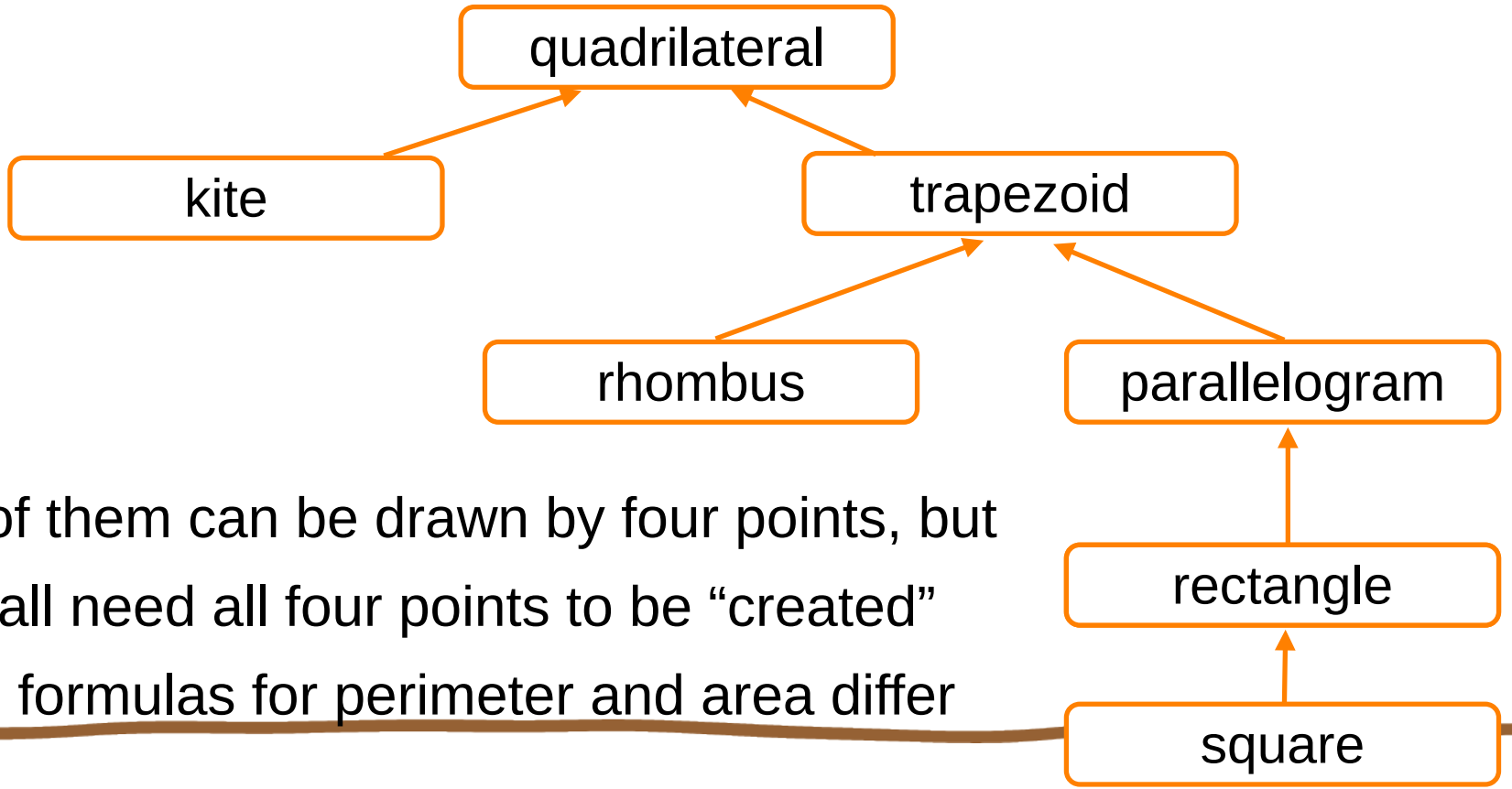
Polymorphism

- We can think of the following inheritance hierarchy:



Polymorphism

- We can think of the following inheritance hierarchy:



- All of them can be drawn by four points, but not all need all four points to be “created”
- The formulas for perimeter and area differ

Polymorphism

- If we decide to leave the calculation of area and/or perimeter to derived from Quadrilateral classes, then we can use
- Virtual functions that give us the ability to define a function in a base class and have a function of the same name and type in a derived class called when a user calls the base class function,
- This is often called *run-time polymorphism*, *dynamic dispatch*, or *run-time dispatch* because the function called is determined at run time based on the type of the object used.

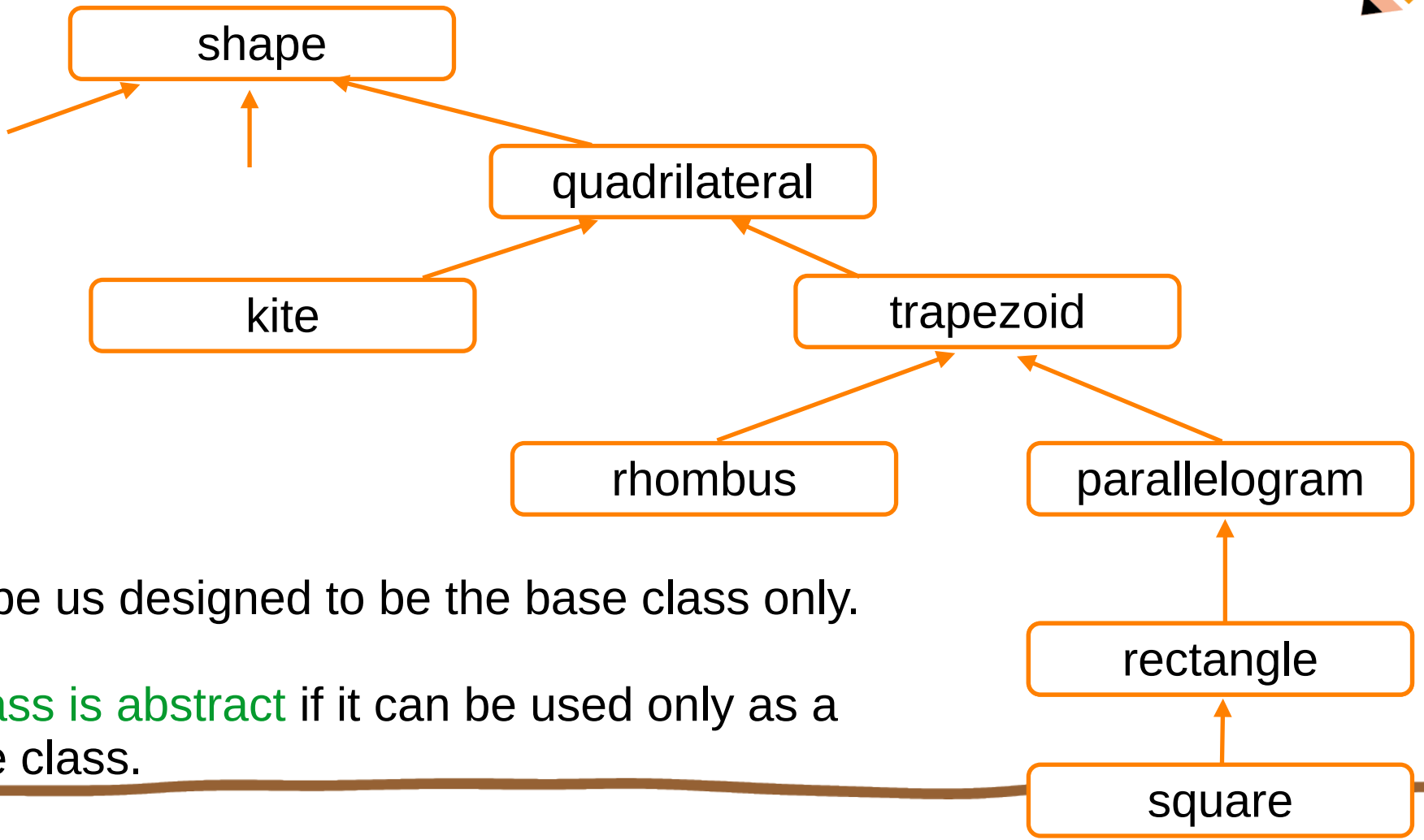
Polymorphism

```
class Quadrilateral
{
public:
   Quadrilateral(const Point& a, const Point& b,
                 const Point& c, const Point& d);

      // .. skipped

   virtual double Perimeter() const; //perimeter
   virtual double Area() const; // area
   //…
};
```

More about graphics



Shape us designed to be the base class only.

A class is abstract if it can be used only as a base class.

More about graphics

```
class Shape {
public:
   void draw() const; // deal with color and draw lines
   virtual void move(int dx, int dy); // move the shape +=dx, +=dy

   // something for line and fill colors

   virtual ~Shape() {} // left for derived classes
protected:
   Shape() {};
   Shape(std::initializer_list<Point> lst);

   void add(Point p); // add p to points
   void setPoint(int i, Point p); // points[i] = p
private:
   std::vector<Point> points; // not used by all shapes
};
```

Resources used for these slides

- slides provided by B. Stroustrup at https://www.stroustrup.com/PPP2slides.html
- Class textbook
- Problem Solving with C++, 7th edition, by Walter Savitch, Pearson
- C++ How to Program, 10th Edition, by Paul Deitel and Harvey Deitel, 2017, Pearson