Polymorphism

- Polymorphism allows the programmer to provide alternate, abstracted views of an object.
  - While this isn't useful if all we have is `Circle`, the reasoning becomes more important with `Square`, `Pentagon`, and `Hexagon` (for example).
  - `Squares` aren't `Circles`, but they *are* both `Shapes`, with `Shape` properties.

Polymorphism

- All of the following are legal code lines, assuming good class definitions.

```
Shape* s1 = new Circle(4);
Shape* s2 = new Square(4);
Shape* s3 = new Pentagon(4);
```

Polymorphism

- Suppose, then, that we have `vector<Shape*> shapeList`, and want to sum up the area of all the stored, referenced Shapes.

```
double areaSum = 0;


for(int i=0; i < shapeList.size(); i++)
    areaSum += shapeList[i]->area();
```

C++ Practicalities

- Note how we're storing each Shape via reference.
  - This is because in creating a by-value variable of Shape, it is attached directly to an inherent Shape instance.
  - For one, this is impossible to instantiate, as Shape has virtual methods.
  - Secondly, that instance would be of *exactly* type Shape – attempting assignment would be on an implied `Shape::operator=()`.

C++ Practicalities

- Note how we're storing each Shape via pointer.
  - In short, use of polymorphism in C++ requires handling objects via their pointers.
  - A pointer of a subclass can always be stored as a variable of its superclass.
  - Technically, through polymorphism, a pointer of a subclass *is a* pointer to an instance of the superclass – just with further specification.

A Quick Note

- At this point in the class, we will *not* be examining full "inheritance."
  - Understanding and being able to use polymorphism is a large-enough issue at this time.
  - The main difference: full inheritance allows for pre-*defined* methods and fields; actual functionality is, well, *inherited*.

A Quick Note

- While our present example used only virtual, undefined methods in the base type (Shape), this is not a strict requirement of C++; more can be inherited.
  - Our present aim is to understand the design goals behind polymorphism, which also affects inheritance in C++.

Polymorphism

- Polymorphism allows for multiple classes to share similar abstract views that may be seen as a single "role".
  - Very distinct, different types sometimes share functionally similar methods.
  - The implementation of these methods may be specific to each implementing class, and is not directly sharable.

Polymorphism Review

- Step 1: creating an abstract base class that *declares* the common methods.
  - One or more methods are declared (but *not* defined).
  - Use the `virtual` keyword!
  - These methods have specifications that should be followed whenever they are implemented.

Polymorphism Review

- Step 2: declaring our classes to be extensions of that abstract base class.
  - This allows instances of our classes to be considered as instances of that abstract base type.

Polymorphism Review

- Step 3: implementing the declared methods of the base class.
  - All undefined methods must be implemented for the class to be instantiated.
  - These implementations should follow the base class's specifications.

Toward Inheritance

- Note that in our original base classes for polymorphism, every declared method was "pure" `virtual`.
  - This reflects interfaces in Java.
  - This allows a base class to ensure it remains abstract and to *force* a base class to implement the method manually.

Toward Inheritance

```
class Shape
{
   public:
      virtual double area() = 0;
      virtual double perimeter() = 0;
}
```

- For this example, there is no one “right” way to implement area; it depends on each *specific* type of Shape.

Toward Inheritance

- There may be some cases in coding, however, when the base class could provide some functionality for its derived classes.
  - There may be *some* specifics better left to each of the derived classes, but with some core features held in common.

Inheritance

- In C++, entire class specifications can be *inherited* from a base class to a derived class.
  - This includes fields and method definitions.

Inheritance

- Inheriting from a class means that the derived class should be considered a "more specific" version of the base class.
  - It inherits all the original specifications and *adds* more of its own.
  - In C++, any (`public`ly) derived class is automatically polymorphic to its base class.

Access Specifiers

- Thus far, we have seen two access modifiers:
  - `public`: declares a field or method is fully visible from any object
  - `private`: places fields or methods on "lockdown," making them invisible outside of the class.

Access Specifiers

- There exists another access modifier:
  - `protected`: declares a field or method is invisible outside of the class, *except* to those inheriting from the class.
  - This can be very useful to extend core functionality in derived classes.
  - One example: providing an empty `protected` method called by the base class in certain situations.

Access Specifiers

- Note that when extending a class in C++, an access specifier is used.
  - This allows the derived class to restrict access to the base class's members.
  - All base class fields and methods will have their access be at least as strict as the given modifier.
  - `protected` inheritance will cause `public` base class methods to become `protected` within the derived class.

Access Specifiers

- Note that when extending a class in C++, an access specifier is used.
  - The implied polymorphism of the derived class to its base will be similiarly restricted.

The `friend` keyword

- It may be desirable for one class to permit another to have special access rights to its members.
  - The solution: the `friend` keyword.

The `friend` keyword

- Declaring another class as a `friend` grants it private-level access to all fields and methods.
  - This only applies for the exact `friend`-granting class.
  - Likewise, only the exact friend stated is granted special access.

Odds and Ends

- To make sure that a derived class is properly overriding (or implementing) a base class method, the `override` keyword may be appended at the end of the declaration's signature.
  - If the `overriden` method does not exist, or is not virtual, a compiler error will result.

Odds and Ends

- To make sure that a derived class cannot *possibly* override a method, the base class may declare a method `final`.
  - Any attempt to override it in a derived class will be marked as a compiler error.
  - Both `final` and `override` appear at the absolute end of a method declaration signature.