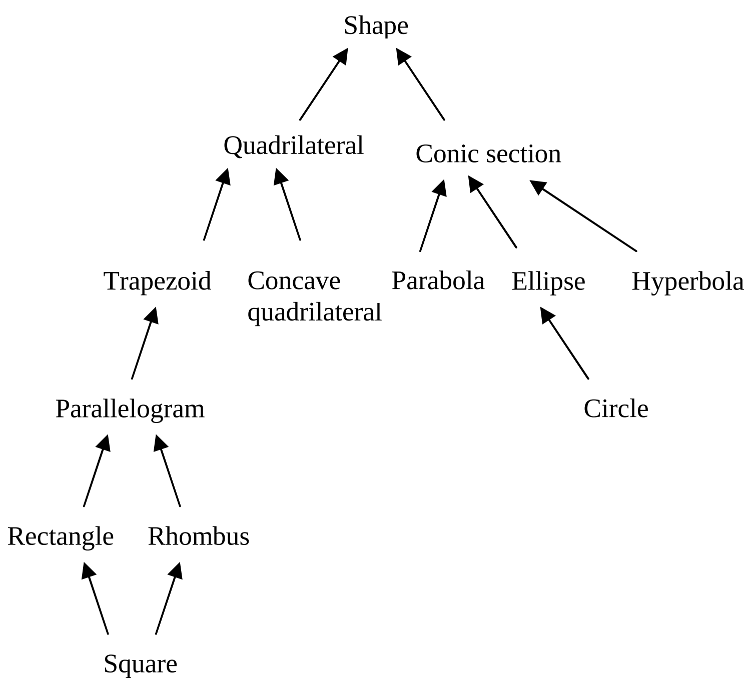
Inheritance and polymorphism

Professors Clark F. Olson and Carol Zander

Inheritance basics

We've seen templates as one method for code reuse. Another important technique is inheritance. One of the important benefits of inheritance is that it allows us to extend a class without modifying (or breaking) the original class.
Single inheritance is where one class is derived from another class. Objects are defined as extensions previously defined objects. The extension inherits all data members and member functions.

Abstractly, inheritance represents an is-a relationship. A rectangle is a shape. A square is a rectangle. We can build hierarchies using this concept. For shapes (the arrow represents an is-a relationship):



The square is an example of multiple inheritance. (The structure is not necessarily a tree.) Note that we draw the arrow from the "child class" or *derived class* to the "parent class" or *base class*. Other terminology is superclass and subclass. Every member of the derived class is also a member of the base class, but not vice versa. Attributes (like private data) that every shape has are inherited by all the descendent classes. This doesn't mean that they each have the same value, but that they have some value for these attributes. For our general shapes, there might not be any attributes that all of the derived classes share. Members of the derived classes can have additional attributes. Quadrilaterals have a center of mass, an area, and a circumference.

Other commonly seen first examples show Mammal as the parent class with child classes such as Human, Dog, etc. Here is a simple example of inheritance in C++. We can first define the base class, which will be Person.

```
class Person {
public:
      string getName() const;
      int getAge() const;
      bool getGender() const;
      void display() const;
private:
      string name;
      bool gender;
      int birthdate;    // Could be days elapsed since Jan. 1, 1800.
};
```

Now we can define derived classes that inherit from Person. For example, every student is a person, so inheriting the attributes (members) from the Person class is reasonable.

```
class Student : public Person {
      float getGPA() const;
      void display() const;
private:
      float gpa;
};
```

The "public Person" after the class name indicates that the Student class inherits from the Person class in "public" way. All public operations in the Person class remain public in the Student class. We could instead use private inheritance in which the public members of Person become private in Student. This is unusual, but there are cases where you don't want people to be able to use the interface to the base class. The default is private inheritance, so it is important to remember the "public."

In the Student class, we have added members that can be used for a student and overridden the display operation. (You *override* an operation if the signature is the same and *overload* it if the signature is different.) If the display operation is used for an object known to be a Student, then the Student's display is used instead of the Person's display. Can it be the case that we don't know whether an object is a student or not? Consider the following code:

```
      Person *p;
      if (someBool)
            p = new Person;
      else
            p = new Student;
      p->display();
```

This is legal code, using a concept called *polymorphism*. Which display routine is used? In this case, it is the Person's display routine, even if p is a Student. However, there are ways to change this. We'll return to this concept in some detail later. Note that the following is illegal code.

```
      Student *p = new Person;      // ILLEGAL
```

A Student is a Person, but a Person is not necessarily a Student.

We can use "partial overriding" by calling the base class method in the derived class method.

```
      void Student::display() const {
            Person::display();
            cout << " " << gpa;
      }
```

Note that data that is private in the base class, cannot be accessed by the derived class. Of course, sometimes we would like to be able to access this data. There are two ways to do this. The first (and safest) way is to use accessor or mutators in the base class. The second is to change the visibility of the data. However, we certainly don't want to change the visibility to "public." There is another visibility that allows derived classes to access the data, but not other classes. This visibility is "protected." So, if we want the Student class to be able to access the birthdate attribute (but not the others), use:

```
protected:
      int birthdate;
private:
      string name;
      bool gender;
```

Note that the default constructor for Student will call the constructor for Person and then perform the default initialization on new data members, such as gpa. If we implement the constructor for Student explicitly, we can still call the constructor for Person if want:

```
Student::Student() : Person() {
      gpa = 0.0;
}
```

(This can be done in the initialization list or in the method code.)

We have to be careful if we want to store an array of People, some of whom are Students. We shouldn't use:

```
        Person list[total];
```

In this case, we lose data from any Student that we try to store, since the array only hold the Person information. On the other hand, we can use:

```
        Person *list[total];
```

A pointer to a Person takes the same amount of space as a pointer to a Student.

This can lead to some problems if we want to perform template operations on the Person array (such as sorting). The comparison would compare the pointer locations, rather than the Person objects. One way around this is to create a PersonPtr class that has a Person * member. You can explicitly write the operator< to follow the pointer and compare the name or age or whatever. The BSTree class doesn't have this problem, because it expects pointers to Objects and dereferences them before comparing them.

Polymorphism

Polymorphism, a powerful tool, is the ability of references and pointers to refer to objects of different types (related by inheritance) and to respond differently, but appropriately and correctly, when a member operation is called. However, we saw previously that if the type of the object is not determined at compile time, then the base class function is used. **Virtual functions** allow us to implement polymorphism. The default operation is to use **static binding**, when the function that will be called is determined at compile-time. This is why the base class function must be used. If the object is a member of a derived class, we won't necessarily know it. For polymorphism to work, we need to have **dynamic binding** that occurs at run-time. We can specify that a member operation should use dynamic binding by declaring it to be **virtual**:

```
        virtual void display() const;
```

The virtualness of an operation is always inherited, so if it is virtual in the base class, it must be virtual in the derived class, *even if it isn't declared as virtual*. (But always use the keyword in children classes for clarity.) In this case, our code above would call the Student display if *p is a Student and the Person display otherwise.

Important notes:

- With dynamic binding, the most specific definition of the operation is used. It may not be in the most specific derived class. (For example, Student might not have a display operation.)
- Constructors are never virtual. We always know what we are constructing at compile time. However, it is common to implement a "clone" function so that we can replicate a collection of objects (using pointers) that derive from a base class correctly.
- Destructors should be virtual if memory is allocated (even if it is static) in any derived class to avoid memory leaks. Otherwise, static binding could call the base class destructor and miss the memory allocated in the derived class.

It is possible for a base class method to be **pure virtual**. This means that the method is not even defined in the base class.

```
        virtual void display() const = 0;
```

Any class with a pure virtual function is called an **abstract class**. The method must be implemented in a derived class for it to be a **concrete class**. Otherwise, the derived class will also be an abstract class. You can never instantiate an abstract class (even if you don't use the virtual functions), but you can have pointers to it. So, if Person was an abstract class (owing to display being pure virtual), this would be illegal:

```
        Person p;
```

However, this would be legal (if display is defined in Student):

```
        Person *p = new Student;
```

Any class derived from an abstract class that doesn't implement all virtual functions is also an abstract class.

More issues:

- Parameter types are always determined statically (at compile-time). Consider the following operations:

  ```
  ostream &operator<<(ostream &out, const Person &p);
  ostream &operator<<(ostream &out, const Student &s);

  Person *p = new Student;
  cout << *p;
  ```

  Which operator is called? It is the Person version. This can be fixed by calling a virtual print function from operator<<.

- If a class is a friend of a base class, it gets access to the inherited attributes. If the base class is a friend of another class, the derived class is not necessarily a friend of that class.
- Using call-by-value with polymorphic data can result in "slicing" of data. If we call a function and pass a Student by value into a Person parameter by value, the call will compile and run, but the parameter will not have all of the Student attributes. In that function, its type is Person.
- Don't make the destructor pure virtual. The derived class destructors need to call the base class destructor. Destructors are a special case in which the base class method is always called.

Casting

Note that to override a function, you need to use the exact same prototype for the function (with one exception). You cannot substitute a derived class for a base class parameter (or vice versa). For example, you can't override:

```
	bool Person::operator<(const Person &p) const
```

with

```
	bool Student::operator<(const Student &p) const
```

The parameter is different, so it is considered to be overloaded, not overridden.

We can implicitly cast from the derived class to the base class, but not vice versa. Let's say that we want to implement a virtual operator< for people. If they are both students, we can order them by GPA.

```
	bool Student::operator<(const Person &p) const {
		const Student &s = static_cast<const Student &>(p);
		return gpa < s.gpa;
	}
```

We have to override the base class using a Person parameter to match the prototype for the virtual function and make the class non-abstract. Inside the function, we have to cast to a Student, since only they have a GPA. What if the parameter is not a Student?

There is also a dynamic casting operation that does run-time checking:

```
	const Student *s = dynamic_cast<const Student *>(p);
```

If p is not a Student, then the dynamic cast will return NULL. This can be used to prevent casting pointers to other objects into pointers to Students. Note that you must turn on RTTI (run-time type information) to use dynamic_cast in Visual C++ (Properties – C/C++ – Language – Enable Run-Time Type Info). This works with references, too. However, C++ doesn't have NULL references, so dynamic_cast throws an exception if you try to call it with an incorrect parameter.

There is one exception to the rule about having the same prototype. You are allowed to return a pointer or reference to a derived class object instead of the base class. This facilitates operations such as clone.

Multiple inheritance

We saw above that we can inherit from multiple base classes. This can be tricky and should be used only when necessary.

```
	class rhombus;
	class rectangle;

	class square: public rhombus, public rectangle {…};
```

One issue that can arise is that both base classes may define a particular method. You need to be able to specify which one to use. For example, if both the rhombus and rectangle class had a method foo, then we could use:

```
	mySquare->rhombus::foo();
```

Summary

Inheritance is a useful tool for extending classes, allowing code reuse, and writing general containers (your BSTree is general to any class that inherits from Object, with some minor modifications). Polymorphism allows objects to have different behaviors depending on the run-time binding of a variable. However, it is important to use virtual functions to ensure polymorphic behavior.