

Section Notes 7

CS51—Spring, 2008

Week of April 6, 2008

1 Inheritance

Inheritance is one of the fundamental principles of object-oriented programming. Inheritance refers to reusing parts of one class in a class that can serve a similar but more specialized function. In theory, inheritance should save programmers time by allowing them to quickly reuse large portions of almost-solutions to a problem.

The syntax for inheritance is simple, after `class classname`, include : `public parent class`

For instance, if you have a `Student` class and wish to develop subclasses of type `Freshman`, you might write

```
class Freshman : public Student
{
    ...
};
```

We sometimes say that inheritance is an “is a” relationship: for example, “a freshman is a student”. Be careful, though, since this language is ambiguous (more on this later).

Recognize that it is valid for a pointer of a parent class type to point to an instance of a subclass type:

```
Student *pstudent;
Freshman someGuy(...);

pstudent = &someGuy;
```

This works because C++ expects `pstudent` to point to a `Student`, and a `Freshman` is a `Student`, so all is well. This concept is powerful because it allows you to construct, for example, a list of students where each student is actually of a different type (freshman, sophomore, etc.).

Note that the opposite situation, assigning a `Freshman*` to point to a `Student`, is not valid. Why not?

1.1 Virtual methods

When a class inherits a method from another class, it is possible to redefine the implementation of that method in the subclass.

However, imagine you have a parent class pointer that points to a subclass instance, such as our `pstudent` above. If we call a method using that pointer, C++ uses the type of the pointer, `Student`, to resolve which method to call. This means we’ll always call `Student` methods, even though we’re actually pointing to a `Freshman` which might have redefined `Student` methods!

Declaring a method to be virtual solves this problem. Calling a virtual method performs a quick lookup at run-time to call an appropriate subclass method depending on what kind of object the pointer actually refers to. The method cannot be fully resolved in advance, since the pointer might point to objects of different subclass types at different times.

In some cases, parent classes are declared in order to specify the interface for subclasses, and we never want an actual instance of the parent class. To indicate that a certain virtual method exists but is not defined by the parent, use the syntax

```
virtual void methodName() = 0;
```

In this case, we call the method a “pure virtual method” and the class an “abstract class”. You cannot create instances of abstract classes; you must define all pure virtual functions to get a subclass that can have instances.

Pure virtual methods and abstract classes are useful for abstracting away an implementation. You can use a pointer to the abstract class to reference objects subclassed from the abstract class and use any of the methods defined for instances of that class, but without the implementation being pre-specified. This is especially useful when the implementation of the method will depend on other aspects of the subclasses that have not been specified in the parent class.

1.2 Data Hiding

1.2.1 public vs. private

The keywords `public` and `private` describe the accessibility of the data and methods of a class. The `public` keyword means that anyone can access data or methods provided by the class. When might this be useful? Subclasses inherit public fields and methods.

The `private` keyword is much more stringent; it allows no access by outsiders.

(Note to Java programmers: You don’t have to preface every method or field declaration with an access specification.)

1.2.2 friends

In the real world, we sometimes tell our friends things that we don’t want most people to know. In programming, it’s often useful to have classes that are allowed to see the intimate details of other classes even when those details shouldn’t be made generally available. The keyword `friend` allows you to specify that a particular class can see the internals of another class. Although the `friend` keyword can be useful, it should be used sparingly. You can also have friend functions, which are allowed to access all fields of a class even though they are not methods.

1.3 Inheritance versus inclusion

Often a class ends up as a member of a related class rather than as a base class or subclass. This kind of relationship goes by a few different names including object composition. It is not always obvious which relationship to pick. For example, is a dictionary a linked list or does it contain a linked list? In the case of the last assignment, `DictList` included `LLNode` as a member variable. Thus, the `LLNode` was related by inclusion. We sometimes refer to this relationship as a “has a” relationship, because a `DictList` has a `LLNode`.

When in doubt, use inclusion instead of inheritance. It gives you more freedom to modify classes in the future. For more detail on this, you might consider exploring an area of software design known as Design Patterns. For more information, look for references on the web or check out one of the many books on the topic.¹ Design Patterns will not play a prominent role in this course, but are an important topic for any aspiring software developer to master!

¹The canonical reference for Object Oriented Design Patterns is the aptly named “Design Patterns: Elements of Reusable Object-Oriented Software” by the so-called Gang-of-Four (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides). Another book that is less of a reference and more of an introduction to get you started is “Head First Design Patterns” by Eric and Elisabeth Freeman.

2 Example: shapes

Let's design a class hierarchy for shapes. We will start with a **Shape** abstract base class. What methods does it have, and what are their contracts? For example, say we decide to provide `getWidth()` and `getHeight()` as part of the **Shape** interface. How should they be documented to make sense for any shape?

Lots of shapes (squares, rectangles, triangles) are polygons, so it might be useful to have a **Polygon** class that inherits from **Shape**. Should this class be abstract or concrete? What does it add to the **Shape** interface?

In a real graphics system, it would be useful to have a compound shape object, which we might call **Scene**. This class is interesting because it includes a list of shapes as its data representation, but it is also a subclass of **Shape**, since all its shapes together act like one big shape. What would be the general strategy for implementing methods like `rotate()` for a **Scene**?

Here is part of the declaration for **Scene**. Let's implement `Scene::getWidth()` to see the power of virtual methods:

```
class Scene : public Shape
{
    private:
        std::vector<Shape*> shapes;

    public:
        void addShape(Shape *s);
        virtual int getWidth();
};

int Scene::getWidth()
{
```

2.1 The parable of Circle and Ellipse

It is difficult to grasp, at first, what exactly inheritance means and when it should be used. In fact, it is now widely believed that inheritance is *overused*. It is surprisingly easy to see an inheritance relationship between two classes where none actually exists.

The classic example of this is **Circle** and **Ellipse**. We know from mathematics that a circle is a special kind of ellipse, so it seems at first that one might inherit from the other if we were to implement them as C++ classes. But in C++ the “is a” relationship between subclass and parent class must be taken quite literally, such that it is difficult to craft an inheritance relationship between **Circle** and **Ellipse**:

- If **Ellipse** is the base class, we might imagine it would have `setWidth()`, `setHeight()`, `getWidth()`, and `getHeight()` methods. Separating out these methods for a class that represents an ellipse because this geometric shape is defined such that these properties can be varied independently. Now, let's compare this with a class that represents a circle. **Circle** cannot provide implementations of these

methods that seem to work right; changing one dimension would cause the other to change, unlike an ellipse. This is because the height and width of a circle are constrained to be identical – remember that a circle is defined by a single radius.

- If `Circle` is the base class, we might imagine it would have `setRadius()` and `getRadius()` methods. `Ellipse` cannot provide a reasonable implementation for `getRadius()`.

The language “implements the contract of” is more accurate than “is a”. A `Freshman` behaves like a `Student` (or so we hope), and therefore implements the `Student` contract. But a `Circle` cannot correctly fulfill the `Ellipse` contract, and vice-versa.

If we insist on implementing `Circle` as a subclass of `Ellipse`, then we need to *weaken* the `Ellipse` contract. In other words, we must document that `getWidth()` is not guaranteed to return the value passed to the most recent `setWidth()`.

3 Strings and Streams

In C, strings were just `char*` arrays. This leads to all sorts of problems with buffer overflows and insecure code—you may recall that the Morris Worm, an early example of a computer worm, exploited buffer overflows in two common programs, `sendmail` and `fingerd`. More generally, using character arrays to store strings is fraught with difficulties and gotchas.

In C++, you should not use `char*` arrays to imitate strings. Instead, you should use the `string` class provided by the standard library. This class encapsulates the concept of a sequence of characters and provides a number of useful methods on such an object. If you have a `string` object, you can call its `length` method to get its length, or `empty` to see if the string contains any characters. In addition, `string` provides convenient operators for common manipulations. For example, `+` will concatenate two strings. You can also add a string or character onto the end of another string using `+=`.

A C++ string can still be initialized using literals:

```
string s = "My string";
```

and indexed using `s[i]`. Essentially, we keep much of the behavior we learned to use with C, but can easily integrate `string` into our Object Oriented way of thinking.

C++ also provides more convenient types for file I/O. The `iostream` library provides `istream` for input and `ostream` for output. These types are somewhat abstract; you can write functions reading from an `istream` without knowing exactly where the data comes from. In fact, it might come from a file `fstream` or an in-memory `stringstream`.

Console I/O is done through `cout`, `cin`, and `cerr`, which are file streams.

Reading is done by “extracting” from a stream using `>>` and writing is done by “inserting” using `<<` (the stream is always the left operand). Streams also have some useful methods, like `eof`, which tells you when you run out of data.

Streams in C++ have a different interface than what we saw in Scheme. Luckily, they are not too complicated so you will find that in relatively short order you can write some sample programs to familiarize yourself with streams – just start with the Console I/O streams that are described above and feel free to ask a TF or post to the Bulletin Board if you have any questions on how to use streams appropriately.