Lecture 7 Notes: Object-Oriented Programming (OOP) and Inheritance

We've already seen how to define composite datatypes using classes. Now we'll take a step back and consider the programming philosophy underlying classes, known as *object-oriented programming* (OOP).

1 The Basic Ideas of OOP

Classic "procedural" programming languages before C++ (such as C) often focused on the question "What should the program do next?" The way you structure a program in these languages is:

1. Split it up into a set of tasks and subtasks
2. Make functions for the tasks
3. Instruct the computer to perform them in sequence

With large amounts of data and/or large numbers of tasks, this makes for complex and unmaintainable programs.

Consider the task of modeling the operation of a car. Such a program would have lots of separate variables storing information on various car parts, and there'd be no way to group together all the code that relates to, say, the wheels. It's hard to keep all these variables and the connections between all the functions in mind.

To manage this complexity, it's nicer to package up self-sufficient, modular pieces of code. People think of the world in terms of interacting *objects*: we'd talk about interactions between the steering wheel, the pedals, the wheels, etc. OOP allows programmers to pack away details into neat, self-contained boxes (objects) so that they can think of the objects more abstractly and focus on the interactions between them.

There are lots of definitions for OOP, but 3 primary features of it are:

- Encapsulation: grouping related data and functions together as objects and defining an *interface* to those objects
- Inheritance: allowing code to be reused between related types
- Polymorphism: allowing a value to be one of several types, and determining at runtime which functions to call on it based on its type

Let's see how each of these plays out in C++.

2 Encapsulation

Encapsulation just refers to packaging related stuff together. We've already seen how to package up data and the operations it supports in C++: with classes.

If someone hands us a class, we do not need to know how it actually works to use it; all we need to know about is its public methods/data – its *interface*. This is often compared to operating a car: when you drive, you don't care how the steering wheel makes the wheels turn; you just care that the interface the car presents (the steering wheel) allows you to accomplish your goal. If you remember the analogy from Lecture 6 about objects being boxes with buttons you can push, you can also think of the interface of a class as the set of buttons each instance of that class makes available. Interfaces abstract away the details of how all the operations are actually performed, allowing the programmer to focus on how objects will use each other's interfaces – how they interact.

This is why C++ makes you specify public and private access specifiers: by default, it assumes that the things you define in a class are internal details which someone using your code should not have to worry about. The practice of hiding away these details from client code is called "data hiding," or making your class a "black box."

One way to think about what happens in an object-oriented program is that we define what objects exist and what each one knows, and then the objects send messages to each other (by calling each other's methods) to exchange information and tell each other what to do.

3 Inheritance

Inheritance allows us to define hierarchies of related classes.

Imagine we're writing an inventory program for vehicles, including cars and trucks. We could write one class for representing cars and an unrelated one for representing trucks, but we'd have to duplicate the functionality that all vehicles have in common. Instead, C++ allows us to specify the common code in a `Vehicle` class, and then specify that the `Car` and `Truck` classes share this code.

The `Vehicle` class will be much the same as what we've seen before:

```
1 class Vehicle {
2 protected:
3     string license;
4     int year;
```

```
5
6  public:
7      Vehicle(const string &myLicense, const int myYear)
8          : license(myLicense), year(myYear) {}
9      const string getDesc() const
10         { return license + " from " + stringify(year);}
11     const string &getLicense() const {return license;}
12     const int getYear() const { return year;}
13 };
```

A few notes on this code, by line:

2. The standard `string` class is described in Section 1 of PS3; see there for details. Recall that `string` can be appended to each other with the + operator.

3. `protected` is largely equivalent to `private`. We'll discuss the differences shortly.

8. This line demonstrates member *initializer syntax*. When defining a constructor, you sometimes want to initialize certain members, particularly const members, even before the constructor body. You simply put a colon before the function body, followed by a comma-separated list of items of the form `dataMember(initialValue)`.

10. This line assumes the existence of some function `stringify` for converting numbers to `string`s.

Now we want to specify that `Car` will inherit the `Vehicle` code, but with some additions. This is accomplished in line 1 below:

```
1  class Car : public Vehicle { // Makes Car inherit from Vehicle
2      string style;
3
4  public:
5      Car( const string & myLicense , const int myYear , const string
           &myStyle)
6          : Vehicle(myLicense, myYear), style(myStyle) {}
7      const string &getStyle() {return style;}
8  };
```

Now class `Car` has all the data members and methods of `Vehicle`, as well as a `style` data member and a `getStyle` method.

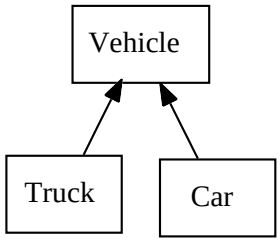
Class `Car` *inherits from* class `Vehicle`. This is equivalent to saying that `Car` is a *derived class*, while `Vehicle` is its *base class*. You may also hear the terms *subclass* and *superclass* instead.

Notes on the code:

1. Don't worry for now about why we stuck the `public` keyword in there.

6. Note how we use member initializer syntax to call the base-class constructor. We need to have a complete `Vehicle` object constructed before we construct the components added in the Car. If you do not explicitly call a base-class constructor using this syntax, the default base-class constructor will be called.

Similarly, we could make a `Truck` class that inherits from `Vehicle` and shares its code. This would give a *class hierarchy* like the following:



Class hierarchies are generally drawn with arrows pointing from derived classes to base classes.

3.1 Is-a vs. Has-a

There are two ways we could describe some class A as depending on some other class B:

1. Every `A` object *has a* B object. For instance, every `Vehicle` *has a* `string` object (called license).

2. Every instance of `A` *is a* `B` instance. For instance, every `Car` *is a* `Vehicle`, as well.

Inheritance allows us to define "is-a" relationships, but it should not be used to implement "has-a" relationships. It would be a design error to make `Vehicle` inherit from `string` because every `Vehicle` has a license; a `Vehicle` is not a `string`. "Has-a" relationships should be implemented by declaring data members, not by inheritance.

3.2 Overriding Methods

We might want to generate the description for Cars in a different way from generic Vehicles. To accomplish this, we can simply redefine the `getDesc` method in `Car`, as below. Then, when we call `getDesc` on a `Car` object, it will use the redefined function. Redefining in this manner is called *overriding* the function.

```
1 class Car : public     Vehicle   { // Makes  Car  inherit   from  Vehicle
2     string style ;
3
```

```
4  public:
5      Car( const string & myLicense , const int myYear , const string
           &myStyle)
6          : Vehicle(myLicense, myYear), style(myStyle) {}
7      const string get Desc () // Overriding this member function
8          { return stringify ( year ) + ' ' + style + ": " + license
             ;}
9      const string &getStyle() {return style;}
10 };
```

3.2.1 Programming by Difference

In defining derived classes, we only need to specify what's different about them from their base classes. This powerful technique is called *programming by difference*.

Inheritance allows only overriding methods and adding new members and methods. We cannot remove functionality that was present in the base class.

3.3 Access Modifiers and Inheritance

If we'd declared year and `license` as `private` in `Vehicle`, we wouldn't be able to access them even from a derived class like `Car`. To allow derived classes but not outside code to access data members and member functions, we must declare them as `protected`.

The `public` keyword used in specifying a base class (e.g., `class Car : public Vehicle {...}`) gives a limit for the visibility of the inherited methods in the derived class. Normally you should just use public here, which means that inherited methods declared as `public` are still `public` in the derived class. Specifying `protected` would make inherited methods, even those declared public, have at most `protected` visibility. For a full table of the effects of different inheritance access specifiers, see http://en.wikibooks.org/wiki/C++_Programming/Classes/Inheritance.

4 Polymorphism

Polymorphism means "many shapes." It refers to the ability of one object to have many types. If we have a function that expects a `Vehicle` object, we can safely pass it a Car object, because every Car is also a Vehicle. Likewise for references and pointers: anywhere you can use a `Vehicle *`, you can use a `Car *`.

3.4 virtual Functions

There is still a problem. Take the following example:

```
1 Car c("VANITY", 2003);
2 Vehicle *vPtr = &c;
3 cout << vPtr->getDesc();
```

(The -> notation on line 3 just dereferences and gets a member. `ptr->member` is equivalent to `(*ptr).member`.)

Because `vPtr` is declared as a `Vehicle *`, this will call the `Vehicle` version of `getDesc`, even though the object pointed to is actually a `Car`. Usually we'd want the program to select the correct function at runtime based on which kind of object is pointed to. We can get this behavior by adding the keyword `virtual` before the method definition:

```
1 class Vehicle {
2     ...
3     virtual const string get Desc () {...}
4 };
```

With this definition, the code above would correctly select the `Car` version of `getDesc`.

Selecting the correct function at runtime is called *dynamic dispatch*. This matches the whole OOP idea – we're sending a message to the object and letting it figure out for itself what actions that message actually means it should take.

Because references are implicitly using pointers, the same issues apply to references:

```
1 Car c("VANITY", 2003);
2 Vehicle &v = c;
3 cout << v.getDesc();
```

This will only call the `Car` version of `getDesc` if `getDesc` is declared as `virtual`.

Once a method is declared virtual in some class C, it is virtual in every derived class of C, even if not explicitly declared as such. However, it is a good idea to declare it as virtual in the derived classes anyway for clarity.

3.5 Pure virtual Functions

Arguably, there is no reasonable way to define `getDesc` for a generic `Vehicle` – only derived classes really need a definition of it, since there is no such thing as a generic vehicle that isn't also a car, truck, or the like. Still, we do want to require every derived class of `Vehicle` to have this function.

We can omit the definition of `getDesc` from `Vehicle` by making the function *pure virtual* via the following odd syntax:

```
1 class Vehicle {
2     ...
3     virtual const string get Desc () = 0; // Pure virtual
4 };
```

The = 0 indicates that no definition will be given. This implies that one can no longer create an instance of `Vehicle`; one can only create instances of `Car`s, `Truck`s, and other derived classes which do implement the `getDesc` method. `Vehicle` is then an *abstract* class – one which defines only an interface, but doesn't actually implement it, and therefore cannot be instantiated.

5 Multiple Inheritance

Unlike many object-oriented languages, C++ allows a class to have multiple base classes:

```
1 class Car : public Vehicle , public Insured Item {
2     ...
3 };
```

This specifies that Car should have all the members of both the `Vehicle` and the `InsuredItem` classes.

Multiple inheritance is tricky and potentially dangerous:

- If both Vehicle and InsuredItem define a member x, you must remember to disambiguate which one you're referring to by saying Vehicle::x or InsuredItem::x.
- If both Vehicle and InsuredItem inherited from the same base class, you'd end up with two instances of the base class within each Car (a "dreaded diamond" class hier- archy). There are ways to solve this problem, but it can get messy.

In general, avoid multiple inheritance unless you know exactly what you're doing.