CS 246 Fall 2016 - Tutorial 8

November 2, 2016

1 Summary

- Class Relationships
- Inheritance

2 Tip of the Week: commands from vi

- Commands in general: when working in vim, you can call any command you would call from the terminal. The command is

  `:!command`

- Calling make from vi: while you can call make as explained above, you can also call make using `:make`. This runs the makefile in the current directory. If an error occurs while compiling, you can hit enter and the vi will jump to the line where the compilation error occurred. If you want to move onto the next error, enter `:cn`.

3 Class Relationships

There are three types of relationships between classes which we typically discuss.

- Composition (owns-a): class A owns an instance of class B. This means that class A is responsible for deleteing the instance of class B when an object of class A is destroyed.
- Aggregation (has-a): class A has an instance of class B. This means that class A is not responsible for deleting the instance of class B.
- Inheritance (is-a): class B is an instance of class A. This means that an instance of class B can be used in any situation where an instance of class A can be used. Note: the converse is not true.

Note: if a class A has a pointer to class B, you cannot know if the relationship is composition or aggregation without looking at the source code (or documentation).

```
class B{
  ...
};

class A{
  B b; // this is composition
  B* b2; // this could be composition or aggregation
}
```

4 Inheritance

Example:

```
class A{
  int a;
 public:
  A(int a): a{a} {};
};

class B: public A{
  int b;
 public:
  B(int a, int b): A{a}, b{b}{};
};
```

In this example, B inherits from A (this is what the ": public A" is for). This means that every instance of B has the fields and methods which an A has.

Note the constructor for the B class. The first element of the MIL is `A{a}` which is calling the constructor for the A portion of the B.

5 Encapsulation and Inheritance

If A has members which are private, B cannot access these fields (as they are private). What are some benefits of an inherited class not having direct access to fields of the superclass? Other people may inherit from our classes and this means they'd have access to ther implementation of the class which breaks encapsulation.

However, we often want to give out subclasses special access to the class. For instance, perhaps, we want to have some accessor methods so that subclasses can access fields in a way that we choose but we don't want to let everyone have access to these fields. For this purpose, we can use the third type of privacy: protected.

Members which are protected can be accessed directly by subclasses but cannot be accessed by the public. Note: you should not make fields private as this also breaks encapsulation.

6 Polymorphism

As previously stated, if there is an inheritance relationship between two classes, an instance of the subclass can be used anywhere the superclass can be used.

This means that each of the following is valid:

```
B b{1, 2};
A a = b;
A* a = new B{3, 4};

void foo(A a);
void foo2(A& a);


foo(b);
foo2(b);
```

Note however, that a B object is larger than an A object (it has an extra field). This means that any time we force a B into an A object, it doesn't fit and the object will be sliced. This could potentially lead to memory leaks.

7 Arrays and Inheritance

Continuing with A and B, consider:

```
void foo(A* arr){
  arr[0] = A{10};
  arr[1] = A{7};
}

B arr[2] = {{1,2},{3,4}};
foo(arr);
```

What happens with this code? Well, it compiles fine as the types match. However, the function foo believes the array which it receives is an array of A's. This means that when we assign a value to arr[1], the value 7 will actually be assigned to the location where 2 is stored.

This means that our data is misaligned and while what we are doing in this case is predictable, this is very dangerous.

Take away: never use array objects polymorphically. If you want polymorphic array, use an array of pointers.

8 Override and Virtual

When working with inherited classes, we will often want to specialize the functions to work differently with subclasses.

```
struct Animal{
  virtual bool fly() const { return false; }
};

struct Cat: public Animal{
};

struct NyanCat: public Cat{
  bool fly() const override { return true; }
};

struct Bird: public Animal{
  bool fly() const override { return true; }
};

struct Penguin: public Bird{
  bool fly() const override { return false; }
};
```

First note that we have declared fly() a virtual. Declaring a method virtual means if we override it in a subclass, we will use the subclass version of the method through polymorphic pointers. If we do not override the method, the definition in the most recent ancestor will be used. For instance, calling fly() on a Cat will return false.

Note: the virtual method will be called when dealing with polymorphic pointers. i.e. `Animal a = Bird; a.fly();` returns false.

Using the keyword virtual, tells the compiler to check that the function is that the function is virtual in a superclass and causes a compiler error if it is not.