CS 246 Spring 2019 - Tutorial 8

July 2, 2019

1 Summary

- Static Members
- Class Relationships
- Inheritance
- Encapsulation and Inheritance
- Polymorphism
- Arrays and Inheritance
- `override` and `virtual`

2 Static Members

- In CS136, you saw the keyword `static` being used to declare that variables or functions are only visible to the file they are defined in. `static` can still be used in this way in C++.
- However, we can now declare members of a class as being `static`. A `static` member is a field or method which can be accessed by every instance of the class and (potentially) globally.
- Static fields **must** be initialized in an external file.
- Suppose we want to implement an increment function for the `Node` class as well as store a value to increment all nodes by as follows.

```
struct Node{
...
static int inc;
static void setInc(int value);
void increment();
...
};
```

- Note that all instances of the class have access to all static members. This means that each node has access to the same value `inc` and `setInc()`. Static methods have access to all other static members.

- You can access a static member through an object of that type or through the class directly:

```
cout << Node::inc << endl;
Node::setInc(5);

Node a{5, nullptr};
cout << a.inc << endl;
a.setInc(3);
```

- A common use for `static` fields is as a counter for the number of instances of the class have been created.
- `static` fields are also useful when we want all objects of a class to share something, but not store it in multiple locations.

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 deleting 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. That is, an instance of class A cannot always be used where an instance of class B can be used.

Note: If a class A has a pointer to an instance of class B, you can not 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 instance of 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 object.

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 the fields of the superclass?
  - Other people may inherit from our classes and this means they'd have access to the fields of the superclass in their implementation of their class.
  - **This breaks encapsulation.**
- However, we often want to give 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 members.
- 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 `protected` 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 an instance of the superclass can be used.
- This means that each of the following is syntactically legal, but not necessarily 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 object into an A object, it doesn't fit and the object will be **sliced**; only the A part of B is copied (if a copy is made) or considered as valid access to members (if a reference or a pointer of type A to the object of type B is made).
  - This "slicing" is called *object coercion*

7 Arrays and Inheritance

- Continuing with A and B, consider the following situation:

```
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 perfectly 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 a polymorphic array, use an array of pointers.

8 `override` and `virtual`

- When working with inherited classes, we will often want to specialize the methods to work differently with different subclasses:

```
// Full example at animals/animals.cc

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

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

struct Goose: public Bird{
    bool fly() const override {
        cout << "THANK MR. GOOSE" << endl;
        return false;
    }
};
```

- Note that we have declared `fly()` as a `virtual` method.
  - 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/references**. This does not work with **objects**.
  - For example:

```
        // What is this line of code actually doing?
Animal a = Bird{};
a.fly();
```

    returns false.
- Using the keyword `override` tells the compiler to check that the method is actually an override of a `virtual` method in a superclass and causes a compiler error if it is not.
  - Although the keyword is not required to override a `virtual` method, *it is extremely highly recommended to prevent hours of debugging a mistake (such as a typo in the function signature).*

9 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. In command mode, the syntax for that 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`.