CS 246 Fall 2013 - Tutorial 6

October 29, 2013

1 Summary

- Singleton pattern
  - `static` modifier
- Visibility
- Object Composition

2 Singleton Pattern

- The Singleton design pattern ensures that only a single instance of a class is ever created
- This can be useful when we have shared resource (say, a database)
- To do this in C++ we require a new modifier, called `static`

2.1 `static`

- `static` can be applied to both fields and methods
- `static` fields and methods are associated with the class and not any particular instance of the class
- `static` methods can only access `static` fields - as it has no implicit `this`
- `static` fields can be accessed by both `static` and non-`static` methods
- We can access `static` fields or methods (outside of the class) by qualifying their name with the classname
  - e.g. `ClassName::FieldName` or `ClassName::MethodName()`
- Let's consider a simple example:

```
struct Burrito{
  static string store;
  string toppings;
  double price;
  static string getStore(){ return store;}
};
string Burrito::store = "Holy Guacamole";
```

- `Static` fields must be initialized somewhere (typically in the .cc/.cpp file)

2.2 Singleton

- So let's implement the Singleton pattern using static (although, we could likely get away using global variables and functions).
- In `Database.h`:

```
#ifndef __DATABASE_H__
#define __DATABASE_H__
#include <string>
struct Database{
    static Database* singleton;
    static Database* getInstance();
    unsigned int users;
    Database() : users(0){}
    void addUser(std::string id);
    unsigned int getCount();
};
#endif
```

- In `Database.cpp`:

```
#include "Database.h"
#include <iostream>
using namespace std;

Database * Database::singleton = NULL;

Database* Database::getInstance(){
    if(singleton) return singleton;
    singleton = new Database;
    return singleton;
}

void Database::addUser(string id){ users += 1;}

unsigned int Database::getCount(){ return users;}

int main(){
    Database * db1 = Database::getInstance();
    Database * db2 = Database::getInstance();
    for(int i=0; i < 10; ++i){
        db1->addUser("foo");
    }
    cout << "db1 count:" << db1->getCount() << endl;
    cout << "db2 count:" << db2->getCount() << endl;
    for(int i=0; i < 10; ++i){
        db2->addUser("foo");
    }
    cout << "db1 count:" << db1->getCount() << endl;
    cout << "db2 count:" << db2->getCount() << endl;
}
```

- Note the qualified names for function definitions and initial singleton value
- Currently, we still have some problems. People could still create their own Databases as they can access the constructor.
- What if we wanted to set a Database adminstrator what might we need to do?
- When does the destructor get called? Better yet, when do we delete the Database object?
- There are several solutions. In class, you've seen (at least) one. What is it?

3 Visibility

- Sometimes we want to restrict access to methods or fields of an object
- This could be due to privacy concerns or to force a particular usage
- In class, you've seen:

- `public`: which allows any one to access the field/method
- `private`: only objects in that class or friends can access the field/method

- Unlike other languages, C++ qualifies `public/private` methods/fields as a section and not on each method or field

```
struct Foo{
 public:
  Foo();
  int getX();
 private:
  int x;
 public:
  Foo(const Foo& f);
 private:
  Foo& operator=(const Foo& f);
};
```

- Recall that we can replace `struct` with `class`
- The only difference is what?
- We can now fix our Database class problem of anyone being able to create an instance using visibility:

```
class Database{
  static Database* singleton;
  unsigned int users;
  Database() : users(0){}
 public:
  static Database* getInstance();
  void addUser(std::string id);
  unsigned int getCount();
};
```

4 Object Composition

- Composition occurs when you embed one object inside another object
  - We call this a "has-a" or a "owns-a" relationship
- "Has-a" typically implies that an object doesn't create or destory its components
- "Owns-a" typically implies that an object does create/destroy its components
- Recall that in UML, we model "has-a" with aggregation (open diamond) and "owns-a" with composition (solid diamond)
- Let's look at an example:

```
class Dog{
  string breed;
 public:
  Dog(string breed) : breed(breed){}
};

class Sheriff{
  Dog rufus;
  string county;
 public:
  Sheriff(string county) : county(county){}
  void setDog(Dog d){ rufus = d;}
};

int main(){
  Dog d ("Corgi");
```

```
  Sheriff rosco("Hazard");
  rosco.setDog(d);
}
```

- But this doesn't compile! Why?
  - Because the initialization list will call the default constructor for Dog but it doesn't have one any more!
  - When we compose objects then we call the default constructor for them if no other constructor is called in the initialization list
- How could we remedy this?
  1. Make a default constructor for Dog

     ```
     class Dog{
       string breed;
      public:
       Dog(string breed) : breed(breed){}
       Dog() : breed("Dane"){}
     };
     ```

  2. Take in a Dog as an additional parameter for the Sheriff constructor

     ```
     class Dog{
       string breed;
      public:
       Dog(string breed) : breed(breed){}
       string getBreed () { return breed;}
     };

     class Sheriff{
       Dog rufus;
       string county;
     public:
       Sheriff(string county, Dog d) : county(county), rufus(d.breed){}
       void setDog(Dog d){ rufus = d;}
     };
     ```

  3. Use the initialization list to provide some default behaviour for the Dog

     ```
     class Sheriff{
       Dog rufus;
       string county;
     public:
       Sheriff(string county) : county(county), rufus("Hound"){}
       void setDog(Dog d){ rufus = d;}
     };
     ```

- Which of these is a better solution?
  - It's **arguable**, however, the third solution involves the least amount of substantial change.
  - In Solution 1, we need to change the public interface of the Dog class (or make Dog and Sheriff friends)
  - In Solution 2, we need to change the public interface of the Sheriff class (by adding a second constructor parameter).
  - Either of those may not be desirable in general.