CS 246 Fall 2013 - Tutorial 5

October 15, 2013

1 Topics

- Preprocessor - #include guards
- Classes
- Constructors

2 Preprocessor

- Recall, that a preprocessor statement is any line that begins with #
- Also recall, Preprocessor statements are evaluated before the code makes it to the compiler
- We can have statements for file inclusion, substitution, and conditional inclusion
- Today, we're just going to focus on #include guards as they will become very important as the course goes on
- Two main goals of #include guards:
  1. Prevent the same code from being included multiple times
  2. Prevent cyclic includes (try to compile `cycle.c`)
- Accordingly, any header (.h) file you write should look like:

```
#ifndef __SOMEHEADER_H__
#define __SOMEHEADER_H__
  ... // function/data/class declaractions
#endif
```

- We'll see some more #include guards in a bit

3 Classes

3.1 The Basics

- Thus far, we've been using structs to organize data
- However, to promote encapsulation and abstraction we need something better
- A class can be seen as a structure with member routines (called methods)
- Some important clarifications:
  - **Structure**: groups together related data
  - **Class**: groups together related data and routines
  - **Object**: is an instance of a class

Structure	Object

`struct Rational{`
`    int numer, denom;`
`};`
`double toDouble(const Rational& rat){`
`    return (double)rat.numer/rat.denom;`
`}`
`...`
`// In C: struct Rational r = {1,2};`
`Rational r = {1,2};`
`cout << toDouble(r) << endl;`
	`struct Rational{`
	`    int numer, denom;`
	`    double toDouble(){`
	`        return (double)numer/denom;`
	`    }`
	`}; // This is a class`
	`...`
	`Rational r = {1,2}; // This is an object`
	`cout << r.toDouble() << endl;`

- Methods take an implicit *this* pointer to the calling object and `toDouble()` could be seen as:

```
struct Rational{
  ...
  double toDouble(Rational* this){
    return (double)this->numer/this->denom;
  }
};
```

3.2 Constructors

- In our original example, we saw that we can intialize structures and objects the same way
- However, this doesn't allow the object to do any meaningful initialization (e.g. open a log file and write to it)
- Constructors allow us to do this
- Constructors are just special methods that are used to perform intialization immediately following allocation
- Constructors take the name of the class and can be overloaded in the usual fashion
- If we don't define the default constructor (e.g. one that takes no arguments) then the compiler gives us one that does some initialization
  - C++ strings are set to null
  - Sub-objects have their default constructor called
  - Pointers and other primitive data are not initialized
- Basically, the implicit default constructor does enough to make an object valid but not necessarily what we expect
- So we should define constructors ourselves:

```
struct Rational{
  ...
  Rational(int n, int d){
    numer = n;
    if (d == 0) denom = 1;
    else denom = d;
  }
};
```

- Once we define any constructor then we lose **every** implicit constructor
- So we might want to define a default constructor for Rational. Left as an exercise.

3.3 const and fields

- Suppose we have the following class definition:

```
#ifndef __STUDENT_H__
#define __STUDENT_H__
#include <string>
struct Student{
```

```
  const unsigned int idNo;
  std::string name;
  double grade;
  Student(unsigned int id, std::string n, double g);
};
#endif
```

- Suppose we have the following definition of the Student constructor:

```
Student(unsigned int id, string n, double g){
  idNo = id;
  name = n;
  grade = g;
}
```

- The compiler is going to complain. Why?
- We need some way to initialize a constant field before we can ever use it.
- C++ allows this with an initialization list

```
Student(unsigned int id, string n, double g) : idNo(id), name(n), grade(g){}
```

- It looks like we're calling a constructor for each of the fields
- In some cases we are (e.g. strings or other sub-objects)
- Note: Initialization happens in declaration order and not list order. Why?

3.4 Copy Constructor

- The copy constructor is another constructor the compiler will implicitly give us
- It is used to copy an object based upon another object
- Typically, this means that the object being copied should not be changed (and so is a `const` reference)
- Suppose we had a modified definition of a Student and we wanted to be able to clone students:

```
#ifndef __STUDENT_H__
#define __STUDENT_H__
#include <string>
struct Student{
  const unsigned int idNo;
  std::string name;
  double* grades;
  int numGrades;
  Student(unsigned int id, std::string n, double gs[], int ng);
  Student(const Student& os);
};
#endif
```

- Then how might we define the copy constructor?

```
struct Student{
... // Assume other constructors defined correctly
  Student(const Student& os)
  : idNo(os.idNo+2000), name(os.name+" Clone"), grades(os.grades), numGrades(os.numGrades){}
};
```

- What's the problem? They share grades! That doesn't seem right.
- What we've done is called a **shallow copy**.
- What we really want is a **deep copy**

```
Student(const Student& os)
  : idNo(os.idNo+1), name(os.name), grades(new double[os.numGrades]), numGrades(os.numGrades)
{
  for(int i=0; i < numGrades; ++i){
    grades[i] = os.grades[i];
  }
}
```

- Now, the two students can have different grades[1].

[1] Potentially. They are clones after all.