CS 246 Fall 2015 - Tutorial 8

November 6, 2015

1 Summary

- Constructors
- Copy Constructor
- Destructors
- Assigment Operator
- Rule of Three
- Visibility

2 Constructors

- By default, 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 basic initialization
  - 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 Student{
  unsigned int idNo;
  string name;
  double grade;
  Student(unsigned int id, string n, double g){
    idNo = id;
    name = n;
    grade = g;
  }
};
```

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

2.1 const and fields

- Suppose we have the following class definition:

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

- 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 Copy Constructor

- The copy constructor is another constructor that the compiler will implicitly give us, if we don't define one
- 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("Clone " + os.name), 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].

4 Destructor

- Destructors are the opposite of Constructors, except you only get one
- A destructor takes the class name, prefixes it with ~, and takes no parameters or return type
- Destructors are used to uninitialize an object at deallocation (e.g. free any heap allocated memory)
- Typically, we use a destructor if we have a non-contiguous object
  - For example, the object has open files, dynamically allocated memory, pointers to other objects, etc
- When is the object's destructor called in the following code:

```
struct Foo{
  int * arr;
  Foo(int n) : arr(new int[n]){}
  ~Foo(){delete [] arr;}
};
int main(){
  Foo x(1);
  Foo y(11);
  Foo *fp = new Foo(20);

  delete fp;
}
```

- What order are the destructors called in? Why this order?

5 Assignment Operator

- The (copy) assignment operator is used to change an existing object's fields to be copies of another existing object
  - More specifically, `this` is not being initialized.
  - `this` already existed and is being modified
- If we don't define an assignment operator then we get an implict one (like the implicit copy ctor) that does memberwise copy
  - Implicit assignment operator basically performs a shallow copy
- Why might we prefer the copy-and-swap idiom to other methods for defining an assignment operator?

[1]Potentially. They are clones after all.

- Allows implicit garbage collection, we don't have to explicitly delete anything
- Reuses code from copy constructor - less chance for errors
- If memory allocation fails, `this` is left in a valid state

- Example: see vector.cc. To implement the deep copy constructor, destructor and assignment operator, compile with the flag `-DBIGTHREE`
- Why do we return a reference to `*this`?

6 Rule of Three

- The Rule of Three states: *If you define one of copy constructor, destructor, or assignment operator then you likely need all three.*
- Why?
  - Typically when we acquire resources (e.g. heap memory, file descriptors) we will need to explicitly handle such resources in some meaningful way (e.g. allocate new memory) when we go to copy objects or destroy objects.
- Do you **always** need to define all three if you define one of them? Why (not)?
  - Not always. Might want to print logging information in a destructor, etc.
  - Might not be sensible/possible to copy the internal data structure and so can only define the destructor.

7 Visibility

- Sometimes we want to restrict access to methods or fields of an object
- Often this is to enforce encapsulation and abstraction
  - We want to force clients to program to the *interface* and not to the *implementation*
  - Allows the underlying implementation to change without affecting client code drastically
- 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 that such a replacement causes is the default privacy of the class (e.g. private for `class` and public for `struct`)