CS 246 Fall 2015 - Tutorial 7

October 30, 2015

1 Summary

- GDB
- Classes
- Constructors
- Copy Constructor
- Destructors
- Assigment Operator

2 GDB

- As we write more complex programs , errors start to crop up all over the place
- Sometimes these errors are easy to identify and somtimes they are hard
- There are a variety of ways to try to find errors
  - A common debugging tool is the print statement
  - Throwing a bunch of print statments into your code that print out variable values can often find the problem
  - ...But not always (especially in concurrent code, where this will often fix the problem)
- Other times we need a tool that allows us to step through the execution of a program
- In first year, you might have used DrRacket's stepper.
- `gdb` is something like that for C/C++
- `gdb` allows you to print variables, set variables, watch variables, set breakpoints, step through execution, etc
- To use `gdb`, we need to compile our program with the `-g` option which provides debugging information to the debugger
  - For example, it keeps variable and function names, line numbers, etc
- Some common commands include:

Command	Description
run [args]	run the program until it crashes or completes
backtrace|bt	print trace of current stack (list of called routines)
print var-name	print value of specified variable
break routine|[filename:]line-no	set breakpoint at routine or line of file
step [n]	execute next n lines (into routines)
continue [n]	skip next n breakpoints
watch var-name	print a message every time var-name is changed
quit	exit gdb

- To start a gdb session you run the command: `gdb <executable name>`
- You will then be prompted to run the program, set breakpoints, etc
- By default, run will run the program until completion or a crash. So it is wise to set breakpoints before you begin.
- See `gdbexp0.cpp`, `gdbex1.cpp`, `gdbex2.cpp` for examples of buggy programs.

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

```
struct Rational{
  int numer, denom;
  ... // constructors
  double toDouble(/*Rational* this*/){
    return (double) numer/denom;
    // Compiler sees:
    //return (double)this->numer/this->denom;
  }
};
```

3.2 Operator Overloading

- Recall, that operators are actually functions and so can be overloaded
- Classes, like structures, can be used in overloaded operators (rational-overload.cpp)

```
#include <iostream>
#include <string>
using namespace std;

struct Rational{
        int numer, denom;
        double toDouble(){
                return (double) numer/denom;
        }

};
Rational operator+(const Rational& lhs, const Rational &rhs){
    Rational temp;
    temp.numer = lhs.numer * rhs.denom + lhs.denom * rhs.numer;
    temp.denom = lhs.denom * rhs.denom;
    return temp;
}
```

```
ostream& operator<<(ostream& out, const Rational &r){
        out << r.numer << "/" << r.denom;
}
```

4 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.

4.1 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?

5 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].

[1]Potentially. They are clones after all.

6 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?

7 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?
  - 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`?