CS 246 Fall 2014 - Tutorial 5

October 15, 2014

1 Topics

- Preprocessor - #include guards
- Classes
- Students Choice

  Streams

  Dynamic Memory

  Other?

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

```
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;
```

Object

```
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 Operator Overloading

- Recall, that operators are actually functions and so can be overloaded
- For example,
  - 2+3 is actually `operator+(2,3)`
  - `string str1=...,str2=...;str1 + str2` is actually `operator+(str1,str2)`
  - `cout << "Hello"` is actually `operator<<(cout, "Hello")`
- This implies that we can define operators for user-defined structures (overloadOperator.cc)

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

struct Rational{
        int numer, denom;
        double toDouble(){
                return (double) numer/denom;
        }
        Rational operator+(Rational &r){
                Rational temp;
                temp.numer = this->numer * r.denom + this->denom * r.numer;
                temp.denom = this->denom * r.denom;
                return temp;
        }
};

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

4 Streams

A2 involved using cin and stringstreams. There were many students who struggled with using these objects to their full potential. As such, we will go over them again.

4.1 IOStreams

- What are the three iostream? cin, cout, cerr
- cin is used for reading from stdin
- cout is used to print to stdout
- cerr is used to prting to stderr

4.2 StringStreams

- A stringstream works exactly like iostream except a stringstream is initialized from a string and looks like

  ```
  stringstream ss("5 10 unicorn elephant");
  ```

- There are istringstreams and ostringstreams which can only be used for input and output respectively. They are useful when you know you will only be reading from or writing to a stringstream as they will cause a program to not compile if you attempt to perform an operation which is not permitted.

4.3 Useful Stream Methods

For input streams:

- stream.fail(): returns true if a read from stream has failed
- stream.clear(): sets fail bit of stream to false
- stream.ignore(): ignores the next char from stream
- stream.peek(): returns the next char in stream, does not remove from the char from stream
- stream.get(): reads the next char from a file
- getline(stream, string): reads the remainder of the current line into string

  Suppose we want to read in lines from cin and return the average of the numbers that occur in each line. (average.cc)

5 Dynamic Memory

- By default (in C++), memory is allocated on the stack
- However, stack allocated memory is invalidated when you leave the scope of the block the memory was allocated in

  ```
  int* foo(){
    int x = 42;
    if (x == 42){
      int y = 84;
    } // y is invalidated
    return &x; // x be invalidated
  }
  ```

- To have memory persist between different scopes (e.g. between functions) then we need to allocate it on the heap
- In C++, operator **new** accomplishes this by taking in a type and allocating the appropriate amount of memory for the given type

  ```
  int * x = new int;
  int * y = new int [10];
  delete x;
  delete [] y;
  ```

- Heap allocated memory is freed using **delete** or **delete []**
- What happens if we delete an array using **delete**?
- Let's consider an example of wanting to increase the amount of memory an array has (readInts.cc):

```
int num;
cin >> num;
int * array = new int[num];
for (int i = 0; i < num; ++i){
        int j;
        cin >> j;
        array[i] = 2*i;
}
int * temp = new int[num];
for (int i = 0; i < 2*num; ++i){
        if ( i < num )
                temp[i] = array[i];
        else
                temp[i] = 0;
}
delete[] array;
array = temp;
```

6 Tip of the Week

- Sometimes we accidentally delete files that we wanted to keep
- In general, deleting a file with `rm` is unrecoverable
- However, on `linux.student.cs` your files are backed up for you
- The `.snapshot` directory in your home directory contains several sub-directories
- They correspond to hourly, nightly, and weekly backups of all your files on the undergrad environment
- You can navigate them to get an older versions of your files
- Note that you should copy (not move or edit) these files into your working directories
  - Otherwise, you risk deleting or otherwise modifying a backup