CS 246 Fall 2016 - Tutorial 4

October 5, 2016

1 Summary

- Dynamic Memory
- Structures
- Operator Overloading
- Preprocessor
- Include Guards

2 Vi Tip of the Week: Yank, Cut, and Paste

- `y`: copy the current line
- `p`: pastes lines which were yanked or deleted after the cursor
- `P`: pastes lines which were yanked or deleted before the cursor
- Visual Mode: you can press `v` to enter visual mode. This allows you to select text by moving up and down the screen. This text can be yanked using `y` or cut using `x` and pasted using p as described above.

  : you can also use ctrl-v to select a vertical block of code and shift-v to select lines of code.

3 Dynamic Memory

- In C++, we use `new` instead of `malloc`, and `delete` instead of free. `new` allocates allocates enough space on the heap for one object and returns a pointer to the allocated memory. `delete` frees the memory stored at the address the variable holds.

```
int * x = new int{5};
...
delete x;
```

- To allocate an array on the heap: `int *arr = new int[10]`. To delete an array on the heap: `delete[] arr`.
- Note: make sure you always use new and delete together, and use `new[]` and `delete[]` together.

4 Classes

- A structure is a collection of data and methods

```
struct Rational{
   int num;
   int den;

void reduce();
};
```

- To access the members of an object: `objectName.fieldName`
- Methods are members as well and are called: `objectName.method()`. `this` is a pointer to the object the method was called on. Methods have access to the members of the object. Members can be accessed directly by calling the member name. (We can also access them through `this` but it is discouraged.)
- To access members through a pointer, the pointer must be dereferenced:
  - `(*objectName).memberName`
  - `objectName->memberName` - syntactic sugar for the first example

5 Operator Overloading

- We can overload many of the built in operators including the arithmetic operators.
- This is often useful when implementing our own structures.
- When overloading operators, we should give our operators logical meanings. i.e. operator+ should do something that resembles addition

5.1 Cascading

- We've gotten used to seeing code that looks like the following:

```
int num = 5;
cout << "The magic number is " << num * num - num << endl;
```

- We now that `num * num - num` execute `num * num` and the result will have `num` subtracted from it. What is happening here is that operator* is being called which returns an int, then operator- is being called and returns the int which is printed.
- Furthermore, operator« returns the ostream it was given which is used later in the expression.
- When we write operators, we want our operators to behave in a similar fashion. This means that our operators will return the result of the operation.
- When writing input or output operators, an istream or ostream will be returned. This will always be a reference to the stream which was input as we do not want to copy streams.
- Let's write operator+, operator*, operator« and operator» for the Rational structure.

6 Preprocessor

- The preprocessor runs before the compiler. It handles all preprocessor directives (any line which begins with #).
- Common preprocessor directives:
  - `#include "file.h"`: pastes the contents of file.h
  - `#define var`: defines a preprocessor macro.
  - `#ifdef var`: includes code if `var` is defined. Must be closed with a #endif
  - `#ifndef var`: includes code if `var` is not defined

6.1 Include Guards

- As you learnt in CS 136, we often want to program in modules which logically separate our code. When we do this, we will often end up including header files in other files so that we have access to functions declared in a module.
- However, we may want to include the same file multiple times in our program which will result in compilation errors.
- To prevent including files multiple times, we are going to setup each header file so that it first checks if a unique marco is defined. If it has not yet been defined, we will include the header file and define the marco. If it has been defined, we won't include the file.
- See `includeGuards` directory for this tutorial for an example.

7 Tip of the Week: Preprocessor Error Messages

While programming, we will often want to debug by printing out statements to check if program is doing what we expect it to do. However, if we forget to comment out the print statment afterwards, our program will not do what we expect.

One simple way around this it to wrap this print statements in a preprocessor macro.

```
#ifdef DEBUG
cerr << "Testing debug mode" << endl;
#endif
```

It can become cumbersome to wrap each statement like this. Another approach is to write a function which will be called for debugging purposes.

```
void debug(const string& s){
#ifdef DEBUG
cerr<< s << endl;
#endif
}
```

This function could be overloaded to handle any built in class or struct you define. You will be able to turn on debugging by compiling with the -D DEBUG flag.