CS 246 Fall 2016 - Tutorial 9

November 9, 2016

1 Summary

- Abstract Classes
- Vectors
- Exception Handling
- Code Review

2 Tip of the Week: commands from vi

- You can do find and replace with vim using: :%s/old/new/g where `old` is the text you want to replace, and `new` is the replacement text.
- Similarly, you can replace with a confirmation using: :%s/old/new/gc

3 Abstract Classes

- The purpose of an abstract class is to allow subclasses to inherit from a base class containing information that is common to all other subclasses, and when you know there shouldn't be any instance of the base class. A class becomes 'abstract' when it has one or more pure virtual functions.
- Note: Your destructors should always be virtual. They must always have an implementation; even if they are pure virtual.
- Why? Because the destructor of the base class is called even when a derived class is destroyed; this is because every derived class possesses the components of the base class. Thus, the destructor of the base must have some implementation (even if the implementation is empty).

Note: Any function that is pure virtual requires its own implementation in the subclasses. We declare a function as pure virtual when we add `= 0;` to the end of it.

```
class B{
  ...
  virtual string hello() = 0; // the function hello is pure virtual
                              // thus making B an abstract class.
  virtual ~B(){};
};
```

```
class A: public B{
    ... // inherits its fields from B
    A* arr;
    A(): arr{new A[5]} {}
    ~A(){delete [] arr;}
    string hello() override{
        return "bonjour";
    }
};
```

Class A inherits from the abstract class B which has a pure virtual destructor. Now, every class that inherits from B has to provide its own implementation of the destructor.

4 Vectors

Vectors make dynamic allocation easier; any time you make an array, you could probably achieve the same result with a vector.

Example:

```
#include <vector>
#include <iostream>

using namespace std;
int main(){
    vector<int> arr; //creating a vector of integers
    int x;
    while(cin >> x){
        arr.emplace_back(x); //continuously adding integers to arr
                             //and increasing the size of the array without
                             //manually allocating more memory.
    }
    for(int i = 0; i < arr.size(); i++){
        cout << arr[i]; //outputting the values in arr.
    }
}
```

The vector class has a wealth of functions to access and manipulate the elements of the vector; knowing them is a matter of looking up the functions in the vector class and how to use them.

5 Exception Handling

Instead of using C-style methods of handling errors, we use exceptions in C++ to control the behaviour of the program when errors arise.

Recall that we catch exceptions with try and catch blocks.

```
try{
    throw 42;
}
catch(...) // ... means 'catch anything'
{
    cerr << "Caught something" << endl;
}
```

We can throw **anything**, and we always catch by reference.
Consider the code below:

```
class BadInput{};

class BadNumber: public BadInput{
    string what() { return "no number given" }
};

int main(){
    try{
        cin >> x;
        if (x < 50){
            throw BadNumber{};
        }
    }
    catch(BadInput& b){}
    catch(BadNumber& b){ cerr << b.what() << endl; } //accessing auxilliary information
                                                     //from object that was caught

}
```

- Question: Which handler would run?
- Note: If we weren't passing objects by reference in the catch parameter, slicing could occur; potentially resulting in missing information.
- Good practice: Throw by value, catch by reference.