CME 211 C++ Exam Review

Tuesday December 3rd, 2019

Usage: this set of notes assumes that you have gone through the notes or went to class at some point. The aim is to take you again through the contents of the course, solidifying the key concepts that you should remember, even after the course!

Contents: (1) a trip through the course in summary; (2) common mistakes observed in the homeworks

1 Types: C style and C++ style

Unlike Python, C++ makes a big deal out of types.

1.1 Primitive data types

- `int` (`short`, `long`, `signed`, `unsigned`)

- `double` (`float`)

- `char`

- `boolean`

In practice, usually the items before the parantheses are what you want to use, unless you're super concerned about speed.

Other prominent data types are `std::string`, and containers like `std::vectors`.

1.2 Containers

1.2.1 C-style containers (e.g. static arrays)

Q: When should you use these?

A: To understand how containers and pointers work. There are better alternatives for C++!

Remember that (static) arrays have a size that is (usually[1]) fixed, and are fussy to use with functions. Here's an array:

```
1    int some_numbers [5];
```

which has a fixed size. This does something similar:

```
1    (int*) some_more_numbers [5];
```

But bEwaRe! The following is an array of **pointers**:

```
1    int* some_pointers_to_numbers [5];
```

Confusing! Indeed pointers add a lot of confusion to C, but the power in pointers is to allow you to return the array from functions.

1.2.2   Returning containers in functions

Here's one way to write a function that returns an array allocated on the heap.

```
1    int* make_me_an_array (int size){
2      int* array = new int [5];
3      // Common pattern: traversing containers like arrays
4      for (int i = 0; i < 5; ++i){
5        array[i] = 0;
6      }
7      return array;
8      // Array "decays" to a pointer
9      // Also good luck, you're in charge of the memory management now
10   }
```

Exercise: what is `new` doing? Recall the heap and the stack. What are the key differences between these two?

You should be a master of arrays. But can I pass containers without pointers, which are highly error-prone? Let's try the following. Will it work?

```
1    // Does this work?
2    int[5] dubious_function (){
3      int array[5] = {0, 0, 0, 0, 0};
4      return array;
5    }
```

---

[1]Recall that some compilers allow variable-length arrays, a non-standard feature

1.2.3   C++: Use a vector!

```
1  std::vector<int> much_better(){
2    std::vector<int> new_vector = {0, 0, 0, 0, 0};
3    return new_vector;
4  }
```

Much less likely to make a mistake when passing vectors into functions.

You should always prefer to use vectors [...] instead of arrays.—Herb Sutter, author of some C++ book.

1.2.4   Exercise: read the code

Here are some super common patterns you should be comfortable with (you should be bored looking at it by now!):

```
1  // Creating zero-vectors of some fixed size
2  std::vector<double> x_range(10, 0.0);
3
4  // Fill with a range
5  for (int i = 0; i < x_range.size(); ++i){
6    x_range[i] = i;
7  }
8
9  // Mapping x to the square of x^2, i.e. squaring each element
10 for (int i = 0; i < x_range.size(); ++i){
11   x_range[i] = x_range[i] * x_range[i];
12 }
13
14 // Sum over a vector
15 double total = 0;
16 for (int i = 0; i < x_range.size(); ++i){
17   total += x_range[i];
18 }
```

1.3   Getting vectors in and out of functions

Our motivation in this review for vectors is passing to and from functions. Well, how do we pass vectors in and vectors out of a function?

```
1  #include <cmath>
2
3  // Takes a vector of ints and returns the square root
4  std::vector<double> sqrt_vector(std::vector<int> input){
5    std::vector<int> output(input.size(), 0.0);
6    for (int i = 0; i < output.size(); ++i){
7      output[i] = std::sqrt((double) input[i]);
8    }
9    return output;
10 }
```

Exercise: can you do it with push_back? What are the differences?

1.3.1 Reference, const reference

Passing a vector as an argument does so by value: in vector's case, each element is copiedIf you copy a vector of pointers, the pointers still point to their original location..

Passing by reference doesn't copy: it refers to the same vector as the one given to it.

```cpp
void mess_up_my_vector(std::vector<double>& target_vector){
  // Let's mess it up real good. These changes will stick!
  for (int i = 0; i < target_vector.size(); ++i){
    target_vector[i] = 0.5*target_vector[i];
  }
}
```

If I don't intend to change the vector I accept as an argument even though I'm taking it as a reference, put a const:

```cpp
void cant_mess_up_my_vector(const std::vector<double>& target_vector){
  // Haha! Now the following won't compile because I promised it should be const.
  // Thus, no unintended changes to target_vector are propagated up
  for (int i = 0; i < target_vector.size(); ++i){
    target_vector[i] = 0.5*target_vector[i];
  }
}
```

Of course, you can take multiple arguments by (non-const) reference to return multiple values. If they are of the same type, you can also return a container such as a vector or a tuple.

2 Classes, constructors, and `this`

When you have a bunch of data (possibly of different types), OOP is a natural way to clump relevant data together, while also bundling in how the data is used (methods).

Here are some common things to think about before starting to code.

- In a method: should I accept something as an argument or use it as a data member?

- Similarly, should I return the computed value? Or should I save it to a data member?

- What information defines the class and needs to be in the contructor?

These questions will outline where the data for each method comes from, and what it needs to output.

Let's do an example (in-class).

2.1   this

A final word on `this`: `this` is a pointer that refers to the current object (aka instance). Here's where it could be handy:

```
1   class SomeClass {
2   private:
3     int x;
4   public:
5     SomeClass(int x) {
6       // Which x am I?
7       x = 3;
8     }
9
10    void print_x(){
11      std::cout << x;
12    }
13  };
```

Here the data member `x` does not get saved. Yikes! We can use

```
1   this -> x = 3;
```

to remedy this problem. Recall `this` is a pointer, and this line is equivalent to

```
1   (*this).x = 3;
```

You may review pointers if you find the above mysterious.

Another way to avoid shadowing and using `this` is to name the input argument and the data differently.

3   Feedback / Tips

- Make sure the code works first—for short programs, functioning code is a high priority

- Good idea to test some snippets of code first before integrating

- Make sure the `argv` is used correctly: `argv[0]` is the program name

- There are many ways to do some things. If it produces the correct result and you believe it to be reasonable, write it down!

- Git push when you do something useful or before you make a big change—gives you a handy reset button

- Warnings are there to help—they might lead you to the source of an error

- When you `open` a file, you must `close` it; if you use `new`, you must `delete` it

- Keep up the good work!