CS170 Week 2 Lecture 1 Notes

Disclaimer: These notes are meant to be read in parallel with Professor Mead's online notes if you have missed class. Topics will not be covered extensively. Here, you will only see the minor details Prof. Mead spoke about which were not on his topic notes.

Pre-Class Mentions/Q&A

- ➔ The submission server is now up and running
- ➔ Send Mead an email if you can't log in
- ➔ No grading will be done with Microsoft compiler (only Clang and g++)
- ➔ You will only need the Microsoft compiler in order to use Dr Memory on Windows
- ➔ (Dr Memory doesn't work with executable files created by Cygwin)
- ➔ There will be no CS170 labs until further notice

Moving From C to C++

(only mentioning the main items that differ for now)

Comments:
" //some comment " is now valid
*" /*some other comment*/ "* is still valid

Parameter Lists:
" int main(void); " is now the same as *" int main(); "* (in C++)
(Both options shown above will express to your machine that main has no input parameters)
it is important to keep this in mind when writing code for both languages (C and C++).

Printing to stdout:
cout is an object (an object is an instantiation of a class).
A class is an encapsulation of functions. (This information is not pertinent to our class yet)
cout does not explicitly require a data type because the compiler knows the type unlike with printf.
Now that we have namespaces, we don't use the .h extension for iostream.
Basically, all that iostream does is include iostream.h.
If the C++ header files have extensions, it's likely because they're from standard C.
cerr – will be used to print to standard error.
Left shift operator " << " is not the bitwise operator in C++, but the insertion operator.
To print using cout – we insert objects (cout) into streams (iostream).
(stream << object)
All CS170 C++ printing will be done in this manner.
cout printing makes it very obvious what you are trying to do unlike printf.
endl vs \n is a style decision and is up to the programmer.

Variable Declarations:
You can now declare variables anywhere in code in C++.
This feature is not necessarily for convenience, but for performance/speed difference.

It's best to declare variables close to where they are used.
Unused variables will still cause warnings in C++.
Errors will occur when trying to use a variable outside of it's scope.
" for(int i = 0; … " is now acceptable code.
Initializing variables in a loop also works in do/do while loops to some extent.
In addition, you should only declare the counting variable outside the loop if you plan to use it outside the loop.

The Literal Char Data Type:
It's still okay to assign a character to an integer.
However, the reverse of this is not okay.

int i = 5;
char c = i;

Still emits a warning in both C and C++ because these are different sizes.

char c = 127; is still legal.
char c = 128; legality of this depends on whether the character is signed or unsigned.

The Const Keyword:
There are two different possible meanings for the const keyword.
1. The value will never change.
2. The value is known at compile time.

" const int i = foo(); " this is legal in itself in C++, while in C the value had to be known at compile time. If you now write *" i = 5; "* this would be illegal because i is constant.
We will now use const when we would have used #define.
While #define ignored scope, const will not.

ODR: One Definition Rule. In file1.c and file2.c foo is defined. Both files
will compile, but the files will not link. (in C)

We won't likely be using any #define's in C++.

The bool Type:
No matter what, boolean type data will evaluate to 0 or 1.
The size of bool is compiler dependent, it could be 1 byte or 4 bytes.
You cannot set the size of bool manually (and you shouldn't care about doing that).

The Structure Tags:
typedef is useful if you are creating a struct in a header file (and you want it to be C/C++ compatible).

~15 minute break~

Namespaces

Namespaces:
Globals are still “not good”.
However, as “nested functions” are not allowed, the global namespace is important to us.
Everything in the C++ standard library is in a namespace so it's important to at least know how to access them.
The same symbol naming conventions apply for C++ as in C.
In defining a namespace, { and } must be included even if the namespace only consists of a single line.
There is no need to prototype namespaces (or anything else that isn't a function).

```
namespace introCppProgramming
{
        int foo = 1;
        int bar = 2;
}
```

This is called opening a namespace. This can be done as many times as desired.
Extern is the equivalent of prototyping a variable.

“ *int Foo();* “ is equivalent to “ *extern int Foo();* “
The default is extern and so we do not use this keyword.

Declarations have no curly braces while definitions do have curly braces.

“ *extern int* ” - declaration
“ *int* ” - definition

Declarations don't take up space (this is why it's safe to include a lot of header files).
As soon as you call a function like printf, this adds more space to your program.
Unless you call a function, it will not effect your space.

Unnamed Namespaces:
It's important to use a lot of helper functions in C++.
You might have 25 helper functions in a project and so to avoid conflict you might put them into an unnamed namespace.

In C, we include <math.h>
In C++, we include <cmath>
Some other include files in C++ are cstdio, cstdlib, cstring and cctype.

If you want to access a global variable or symbol you can use the scope resolution operator.
There is no way to access a symbol in an unnamed namespace if you also have it in the global namespace. All helper functions should be in unnamed namespaces.