Session 2

Review

main function

Every executable has an entrant function called main, which is invoked when executing the program. It has two valid forms. The first form is:

```
int main() {
    // bla bla bla
    return 0; // 0 indicates that everything is peachy
}
```

And the second form is:

```
int main(int argc, char* argv[] ) {
    // bla bla bla
    return 0;
}
```

In the second form, as you can see, the main function takes two arguments. They are:

- int argc: the number of arguments that the executable is called with.
- char* argv[] : an array of c style strings representing those arguments. (you might also see the second argument written as `char** arvg`. Its effectively the same thing and we will get into that when we discuss pointers. . .)

By the way, the type signatures are required, but the names of the two arguments can be anything. By convention, they are *argc* and *argv*. You should probably stick to this naming, because that is what people expect.

You can use either of the two forms for main, although I recommend using the former, simpler one, unless you need to access the calling parameters.

Data Types

C++ is a strongly typed language. All variables have an explicit type which cannot be changed at runtime, unlike python. When you declare a variable, you prefix it with its type name. And it *never ever changes*. That is a big change from Python, so soak it in.

The atomic data types are (off the top of my head):

- bool
- char
- wchar
- short

- int
- long
- long long
- float
- double

For each of the int types , there exists an unsigned type, which does not allow for negative numbers.

short int, int, long, and long long all belong to the family of integers. They differ in terms of the range of integers which they represent/can hold. They can each be spelled explicitly (ie short int, long int)

float, double belong to the family of real numbers. They too differ in terms of the range that each can hold.

Namespaces

C ++ allows you to wrap code in a namespace in order to organize and protect symbols. Namespaces are roughly equivalent to python's nested module definitions.

```
import os
os.path.dirname()

def dirname(foo):
    print "dir dir dir", foo

os.path.dirname("/foo/bar")
```

Namspaces are defined like so:

```
namespace foo {
    void bar() {
        std::cout << "I am in a namespace" << std::endl
    }
}
```

You reference symbols in a namespace by prefixing them with their namespace, followed by `::`

```
foo::bar();
```

You can also either selectively import a symbol into the current namespace:

```
using foo::bar;
bar();
```

Or import everything in a namespace into the current namespace:

```
using namespace foo;
bar();
```

Why am i telling you this now? There are many useful libraries which live in the `std` namespace. We will take a look at a couple of them, but first we need to learn how to include them in our code.

CPP Preprocessor: including other code

All commands prefixed by a pound symbol are preprocessor directives. The preprocess runs as the first step in compilation. One of the primary roles of the preprocess is to include other files into cpp files. This is generally analogous to python's import statement. We will get into what is going on here a bit more later.

There are two forms which we are interested in:

```
#include <LIBRARY>
#include "myHeader.h"
```

The former is used to include headers for compiled libraries and the latter is used to include non-precompiled headers, generally in your own project. The former searches in predefined locations, as well as locations explicitly specified via compiler flags. The latter ("") first searches in the directory in which the directive appears, and then degrades to the former (<>) behavior, in the event that the header file is not found.

Useful Libraries

There are a ton of useful libraries which ship with c++. Let's look at two of them in an example:

iostream and string.

iostream includes a number of useful operators for reading and writing streams. Three common operators are:

- cout : pronounced, see-out, this operator works in conjunction with the << operator to print streams to stdout.
- endl : pronounced just like it looks, this operator is just an os agnostic newline.
- cin : pronounced see-in, this operator works in conjunction with the >> operator to pipe data from stdin to a variable.

All of these operators live in the `std` namespace, so you need to prefix them with `std::` when using them.

```
#include <iostream>

int main() {
    //print hello world to standard out followed by a newline
    std::cout << "hello world" << std::endl;

    std::cout << "how many donuts do you want? ";
    int donut_cnt;
    cin >> donut_cnt;
    std::cout << "you wnat "<< donut_cnt << " donuts " << std::endl;

    return 0;
}
```

Next up is the string class. You are in luck, because it works a lot like python's string class. Furthermore, it exists, which is a bonus, because it didn't in c. Thanks c++.

```
#include <iostream>
#include <string>
#include <algorithm>

int main() {
    std::string name{"Freddy"};
    std::cout << "hello " << name << std::endl;
    std::cout << "I am "<< name.size()<<" characters long" << std::endl;

    // if you have c++11, it is easy enough to do things like capitalize
    for (std::string& c: str) c = toupper(c)

    return 0;
}
```

For more on std::string, (click here)[http://en.cppreference.com/w/cpp/string/basic_string]

Declaration, Initialization, Definition, and Assignment

Let's talk about terminology, because its fun.

Declaration

When you name a variable and define its type, that is a declaration.

Example

```
int foo;
```

Initialization

Assigning an initial value to a variable.

Example of Declaration and Initialization

```
int foo = 0;
int bar{22}; // uniform initialization
string name("fred");
```

Assignment

Assigning a new value to a variable is achieved thusly:

```
int foo;
foo = 7;
foo = 12
```

Definition

In the case of functions, classes, and structs, we can differentiate between declaration and definition. If you provide a definition for your function, class, struct, you are providing an implementation...

Functions

Functions in python and c++ are very similar. In python, to declare a function, you do so using the *def* keyword:

```
def foo(bar):
    """
    do something
    """
```

C++ has no such keyword. Instead, the compiler recognizes functions by their *shape“. Additionally, when you define a function in C++, you are responsible for defining the explicit type, if any, that the function returns, as well as the types of all of the parameters, if any.

Basic Function Declaration

All function declarations look like this:

```
<return type> <function name> ( <parameters>);
```

Examples

```
void greet(std::string);
int addtwo(int);
```

So why would you split up the declaration and the definition? There are a couple of more advanced topics where this comes into play. Without going into two much detail:

(1) It is common practice to split up work into header files (.h or .hpp) which contain declarations for data structures, and implementation files (.cpp) which contain implementations. It allows you as an author to expose an interface to a client but keep the implementation hidden. We will look at this later.

(2) You sometimes have to make the compiler aware of the shape of a function, class, or struct, in a different location from its implementation, so that other components may reference it.

Example foo.cpp:

“‘

#include

using namespace std;

int main() {
foo();
return 0;
}

void foo() {
cout << “FOOOOOOO” << endl;
}
“‘

If you try and compile foo.cpp, you might be surprised, based on your python experience, to learn that the compiler cannot find foo. I mean its right there in the same file, but the compiler is too dumb to find it.

You have to perform something called *forward declaration* in order to get this to work:

```
#include <iostream>

 using namespace std;
```

```
// forward declare foo
void foo();

int main() {
   foo();
   return 0;
}

void foo() {
   cout << "FOOOOOOO" << endl;
}
```

Function Definition

If you go a step farther and provide an implementation, you would simply be adding the guts of the code to the end.

```
<return type> <function name> ( <parameters>){ <implementation> };
```

Examples

```
void greet(std::string name) {
    std::cout << " Hello " << name << std::endl;
}

int addtwo( int num ) {
    return num + 2;
}
```

References - part 1

In c++, you can create a reference or alias to a variable by using the `&`. This is commonly used in function declarations for two reasons:

(1) avoid making a copy of a variable because it is large

(2) allow a function to mutate a parameter variable. (remember, functions are pass by value in C++)

Note :
The ampersand has a second use, which is to take the address of a variable. We will go over this soon when we cover pointers...

The syntax for declaring a reference to a variable looks like this:

```
<type>& <name>;

int bar = 1;
int &foo = bar;
```

The syntax for declaring a reference parameter in a function looks like this:

```
<return type> <name>( <type>& <name>,...);

void foo(string& name);
int add2(int &a);
```

Here is a simple main function:

```
using namespace std;
int main() {
    string aname = "Ralf";
    cout << "before " << aname << endl;
    foo(aname);
    cout << "after " << aname << endl;

return 0;
}
```

Observe the difference between this:

```
void foo(string name) {
    name = "foo";
}
```

And this:

```
void foo(string &name) {
    name = foo;
}
```

Be careful. The following is not a reference:

```
int bar = 1;
int foo = &bar;
cout << foo << endl;;
```

But this is:

```
int bar = 1;
int& foo = bar;

cout << foo << endl;
```

Pointers - part 1

Let's talk about pointers. Python doesn't have an analog. This is new stuff. The concept is pretty simple though, even if the syntax is weird in some situations.

Recall for a moment the idea of a type. A type has a size and a meaning. The size is the size in bytes that the type takes up, and the meaning is the representation of those bytes.

So we know how large a variable of a particular type is, but *where* is it? We talked previously about the heap and the stack - two different areas of memory, but we didnt talk about how we find individual variables.

Addresses

That is where the address comes in. The address of a variable is roughly the bit that it starts at, when counting from zero all the way up to the largest bit of memory physically crammed into the computer you are sitting at.

Now, truthfully, this is not quite accurate. There is a level of indirection or two between the physical location in memory and the logical location you deal with when writing a program. However, suffice it to say, as far as you are concerned, every variable you define lives at a particular memory address provided to you by the operating system.

So really, you can think of variable names as convenient labels for memory addresses. Each variable represents a memory address. Thankfully.

Ampersand

And you can actually ask for that address by using our new friend the ampersand. Unlucky for you, the ampersand is overloaded to represent a reference in some contexts, and an address in others.

```
int foo = 1;
// we all know by know what this will print out
cout << "foo " << foo << endl;

// to print the address of foo
cout << "foo address " << &foo << endl;
```

If you prefix a variable with an ampersand operator, it will return the address of the variable in question. Beware, this is distinct from using the ampersand as part of a declaration. In that case, the ampersand means reference!

Asterisk

The ampersand is only have the story when it comes to pointers. You know how to retrieve an address from a variable, but how do you hold on to an address to a variable?

In comes the asterisk. In a declaration, if you prefix the variable name with an asterisk, that means pointer.

```
int foo = 1;
int *bar = &foo;
```

In the preceding example, we first declare foo as an int and initialize it to the value 1. We then declare bar to be a *pointer to int* and initialize it with the address of foo. By the way, it makes no difference whether the asterisk is glombed onto the end of the type or the beginning of the variable name; folks roll both ways.

```
int* foo;  int *bar;
```

So know you know how to define a pointer and initialize it (and set it for that matter). If you print foo from the example above, what do you think you get?

```
int foo = 1;  int *bar = &foo;  cout << "what does bar look like?
" << bar << endl;
```

That gobblygook is an address. Addresses are awfully important, but you rarely want to display them. You are usually interested in the data living at the address you have. So, how do you get at that data? Our friend the asterisk...

```
int foo = 1;  int *bar = &foo;  cout << "foo is " << *bar <<
endl;
```

Recall that I said a variable is a name for a memory address, and that a type is both an indication of size (number of bytes) and meaning for a variable. (for example `char foo` tells the compiler that there is a thing called foo which will be 1 byte long and named foo. When you use foo, the compiler knows to go to that address, take the bit pattern starting at the address it gave foo, and ending 1 byte past that, and interpret that as a character.)

Rather than a string or a number, a pointer variable's contents is an address. And an asterisk in front of a variable in an expression is basically saying:

Go to this address and peak inside. You will find another address. Go to that address, and use its contents.

Get it?

By the way, we are not limited to having a single indirection. We can have a pointer to a pointer, and a pointer to a pointer to a pointer. Why would we want to do that? There are reasons but they are relatively rare.

“‘

int bar = 2;

int *foo = &bar;
int **foof = &foo;

cout << **foof << endl;
“‘

Why all this pointer stuff?

Remember when I told you (like a few sentences ago) that a variable is just a nice name for a memory address? Not all memory address have names. That is the idea behind dynamic memory allocation. You can ask the computer for a certain amound of memory and it will return a starting address to that memory. You store it with a pointer. And, when you are done with it, you tell the computer to clean it up. More on that later....

const keyword

In c++, we have a *const* keyword which can appear in function parameter lists, return declarations, and after method names to signify the fact that the user won't be mutating said data.

The following compiles:

```
void printfoo(const std::string& foo) {
    cout << foo << endl;
}
```

But this errors, because we attempt to set foo after declaring it as *const*.

```
void printfoo(const std::string& foo) {
    foo = "bla";
}
```

One of the many reasons why this is important, is that it is common in c and c++ to use mutable parameters to change the states of variables, rather than returning new variables, especially when the function needs to return multiple variables.

eg

```
void updateName(string& first, string& last) {
    first = "Mr." + first;
    last = last + " Sr.";
}

int main() {
    string first{"Fred"};
    string last{"Flinstone"};
```

```
    updateName(first, last);
    cout << first << " " << last << endl;

    return 0;
}
```

For the most part, const is pretty simple. However, when it comes to pointers, things can get a bit hairy. That's because you can declare a const pointer, a pointer to const, or a const pointer to const. WTF????

1. const pointer - the const appears after the asterisk. It denotes the fact that the pointer's address may not change. EG `int * const foo;`
2. pointer to const - the const appears before the asterisk. It denotes the fact that the variable's value pointed at via the pointer may not change. EG `const int* foo;` or `int const *foo;`
3. const pointer to const - the const appears both before and after the asterisk. This pointer cannot be repointed and the value pointed at cannot change. EGs `const int * const foo;` or `int const * const bar;`