Chapter 2

Flow Control

C++ and Python have very similar flow control mechanisms, however they have different syntax.

Conditionals

In python, conditional are formed thusly:

```
if <expression> :
    <body>
elif <expression>:
    <body>
else:
    <body>
```

For example:

```
if foo == 1:
    print "foo is one"
elif foo ==2 :
    print "foo is two"
else:
    print "foo is bigger than two:
```

In C++, this translates into:

```
if ( <expression> ) {
    <body>;
} else if (<expression>) {
    <body>;
} else {
    <body>;
}
```

That Python example looks like this in C++:

```
if (foo == 1) {
    cout << "foo is one" << endl;
} else if (foo ==2) {
    cout << "foo is two" << endl;
} else {
    cout << "foo is bigger than two" << endl;
}
```

Just as in Python, both the *else if* and the *else* are optional. In C++, the body of each conditional consists of one or more statements, each terminated by a semicolon. Because each conditional is wrapped in curly braces, each conditional forms its own *local scope*. If you define variables within the scope, they are not visible outside of it. Much like Python.

Ternary Form

Python and C++ both have a ternary expression form. In python, it looks like `<truthy value> if <expression> else <falsy value>`. For example:

```
x = 0
y = 1`
z = 12 if x == y else 42
```

In C++, the ternary is formed like so `(<condition>) ? <truthy value> : <falsy value>`. For example:

```
int x = 0; int y = 1;
int z = (x==y) ? 12 : 42;
```

Switch

C++ has an additional flow control statement which Python lacks - The `switch` statement. It looks like this:

```
switch (<integral expression>) {
    case <value>:
        statements;
        break;
    case <value...N>:
        statements;
        break;
    default:
        statement;
 }
```

Notice that the switch can only deal with expressions that evaluate to integral types (that means integers and enums). Also, notice the use of the `break` keyword at the end of each case. If you forget the `break` statement, C++ will perform a *fall-through*, which just means it will move on to the next case statement body and evaluate it, and so on until it hits a break, or the end of the switch. Many a bug has been introduced by forgetting the `break` statement. You have been warned!

For completeness sake, here is a working example:

```
int foo=2;
switch(foo) {
    case 0:
        cout << "foo is zero" << endl;
        break;
    case 1:
        cout << "foo is one" << endl;
        break;
    case 2:
        cout << "foo is greater than one" << endl;
        break;
    default:
        cout << "unexpected foo" << endl;
}
```

Looping

C++ has a number of different looping constructs, which it inherits from C.

for

The `for` keyword kicks off the bread and butter looping form in C++. This will seem a bit verbose to Python programmers. `for` has three components, separated by semi-colons - a counter variable initialization, a comparison, and a counter increment. For example, to print out the numbers zero to ten:

```
for(int x=0; x<10; x++) {
    cout << x << endl;
}
```

This would result in the following:

```
0
1
2
3
4
5
6
7
8
9
```

In case it isn't obvious, the initialization portion of the *for* loop is only executed the first time through the loop, the conditional section is applied each iteration, and the counter is applied at the end each of aach iteration. Also note that the counter portion can define any stride it wants; it isn't limited to incrementing by 1, or incrementing at all, however that is the norm.

That same code in Python is much shorter:

```
for x in xrange(10):
    print x
```

while

The while form defines a test which it executes upon each iteration through the loop. If this test fails, the loop terminates. for example:

```
int x=0;
while(x < 10) {
    cout << x << endl;
    x++;
}
```

do while

C++ has yet another syntactic looping form. This differs from the *while* form in that it applies its test at the tail end of each iteration of the loop, as opposed to the beginning. For example:

```
int x = 0;
do {
    cout << x << endl;
} while(x<1);
```

range based for

C++ 11 introduced a new looping syntax for compliant containers (and all containers in the standard library are compliant). It will appeal to Python programmers, as it resembles Python's `for..in` syntax. It looks like this:

```
for( <var> : <container var> ) {}
```

As an example:

```
vector<int> y = {1,2,3,4,5,6};
for( int x : y) {
    cout << x << endl;
}
```

I realize that we have not talked about vectors yet, but for now, you can think of the line starting with `vector` as defining a list of ints. Hopefully, that makes sense.

A Couple of Useful Library Components

There are a ton of great library components which ship with C++. Let's look at two of them in an example:

iostream

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;
}
```

string

Next up is the string class. Unlike Python, dynamic strings are not built into the language; they are part of the standard library. C++'s std::string class works a lot like python's string class.

```
#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. You can declare a variable without explicitly giving it an initial value.

Example

```
int foo;
```

Initialization

Assigning an initial value to a variable is known as initialization. It is common to declare and initialize variables at the same time, but it is not required.

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…

More on Functions

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 <iostream>

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 kernel for a certain amount of memory and it will return a starting address to that memory. You store the memory address with a pointer, and you access the value at that address by *dereferencing* the pointer. And, when you are done with the memory, 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;`