Chapter 1

Introduction

C++ is a strongly typed, compiled, high performance language, which evolved from c. It shares c's syntax, as well as c's ability to interface fairly directly with hardware. It can be used to write very "close to the metal" code, as well as highly abstracted code. C++ places more responsibilities on the developer, and it is likely that the two areas which will be the most alien to you, the python developer, are memory management and strict typing. However, the big payoff in taking these responsibilities on is the ability (notice i didnt say guarantee) of producing high performance code.

structure of a C++ program

Like a python program, a C++ program can consist of one or more files. And like a python program, a C++ executable ultimately has a single entry point. In C++'s case that entry point is a function named **main**, of which there can be only one. But before we get much further, let's jump right in and write our first C++ program.

Create a file called *hello.cpp* with the following text in it:

```
#include <iostream>

int main() {
    std::cout << "hello world" << std::endl;
    return 0;
}
```

Now let's compile it on the command line. Unlike Python, which automatically compiles py ascii files into pyc bytecode when run, in C++ you have to explicitly compile and link your code to produce an executable. Every time you make a change to your source code, you have to recompile and relink. For this trivial example, we are going to use our compiler to handle all of this in one go. Depending upon your platform and compiler, the command will be different.

Linux

In Linux, the default compiler is called gcc. Gcc has support for a wide variety of languages, including C and C++. Gcc provides a compiler driver called g++, which is pre-configured to work for C++. It is effectively an alias for `gcc -xc++ -lstdc++ -shared-libgcc`, which won't make complete sense to you for a bit, so don't worry. Anyway, we are going to use g++ to compile our code. In a shell, type the following

```
g++ hello.cpp -o hello
```

Windows

Assuming that you have Visual Studo and the optional C++ Components installed, or the Microsoft Visual C++ Build Tools, open a developer shell (there should be a link in your Visual Studio folder) and type:

```
cl /EHsc hello.cpp /out:hello.exe
```

Here, cl is the name of the compiler. /EHsc is a flag instructing the compiler on how to handle exceptions. And /Out:hello.exe is a flag instructing the compiler on what to name the executable (Technically this isnt necessary for this example, as cl takes the name from the first cpp file supplied.)

Mac

Assuming that you have XCode installed, along with the developer command line tools, you should be able to use clang from the command line.

```
clang++ hello.cpp -o hello
```

Results

Tada, you should have an executable called hello, which will greet the world when run. Even with this trivial program, we have a lot to talk about. So let's get started. First, we are going to write an analogous Python script to help explain what we just did. So, create the following script, called **hello.py**:

```
#/usr/bin/env python

if __name__ == '__main__':
    print "Hello World"
```

Looks like the Python weighed in a bit lighter, eh? Sorry to say, but that will be a common theme in your C++ career; C++ is a bit more verbose than Python. Anyway, let's move on and take a look at some basics of C++:

Data Types

Python

As you may recall, Python has a number of built in data types.

- bool
  - True or False

- int
  - plain integers
- long
  - long integers
- float
  - floating point numbers

- complex
  - having real and imaginary components
- str
  - basic strings
- unicode
  - unicode strings

C++

C++ is a strongly typed language. Each and every variable has an explicit type which cannot be changed at runtime. 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 listed below. It is important to note that their actual sizes are architecture and sometimes compiler specific. I have listed their common sizes for x86:

- bool
  - true or false
- char
  - single characters, defined using single quotes ('a'). 1 byte
- wchar
  - wide characters. Compiler specific and can be as small as 1 byte. For C++11, w have char16_t and char32_t for 16 an 32 bit unicode.
- short
  - small integer. 16 bits at a minimum.
- int
  - normal integer 16 bits at minimum. Usually larger than short, and definitely no smaller.
- long
  - large integer. No smaller than int. At least 32 bits
- long long
  - very large integer. No smaller than long. At least 64 bits
- float
  - real number. 32 bits
- double
  - real number. Precision not less than float. 64 bits .
- long double

  - largest real number. Precision not less than double.
- void
  - no storage. represents nothing
- nullptr
  - null pointer. represents an unassigned pointer

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.

Statements

In python, statements end in newlines. In C++, statements end in semi-colons. That means you can split statements over multiple lines without a problem.

In Python you would have to do this:

```
foo \
=1
```

But in C++ you simply type this:

```
int foo
= 1;
```

Braces - Lots of Braces

C++ is a descendant of C. As such, it uses braces to define scope. This stands in stark contrast to Python, which uses indentation to differentiate scope. So, at the risk of repeating myself, you wrap blocks in curly braces to define a scope, instead of indenting. Formatting is strictly for readability. Lke Python, variables defined in inner scopes are not visible to outer scopes. However, variables defined in outer scopes are visible in inner scopes.

```
{
    int f = 1;
    int g = 2;
    {
        f =2;
        int g = 4;
        cout << f << " " << g << endl;
    }
```

```
    cout << f << " " << g << endl;
}
```

In the preceding example, we have an inner and outer scope. In the outer scope, we define two variables, “f” and “g”.
In the inner scope, we assign “f” a new value, and we create a local variable “g” which masks “g” in the outer scope. When we exit the inner scope, “f” retains the assignment from the inner scope, but “g” does not, because the “g” defined in the local scope is a different “g”. While you don’t quite no enough to verify this yet, you soon will. (if you are wondering what cout, <<, and endl are about, stay tuned.)

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
    """
    print "foo",bar
```

C++ has no such keyword. Instead, the compiler recognizes functions by their *shape*. When you define a function in C++, you are responsible for defining the explicit type of the value, if any, that the function returns, as well as the types of all of the function’s parameters. You do this by prefixing the function name with the name of the return type. If the function does not return a type, you prefix it with *void*. Like Python, C++ function names ar followed by parens enclosing any function parameters. However, in C++ these parameters are also prefixed by their types. Like Python, these parameters may also define default values using the same syntax as Python. For example:

```
bool foo(string bar) {
    cout << "foo " << bar << endl;
    return true;
}
```

As you can see, the above example function returns a boolean value. It is named “foo”, and it has a single parameter, named “bar”, which is of type *string*.

includes, The Modules of the C++ World

In C++, like in Python, you can split your implementation of a program up into multiple files. And like, Python, you can leverage existing libraries to do much of the low level work for you; you are not stuck writing everything from

scratch, and much of what is considered modern C++ is actually provided by the Standard Library, not the language itself (although the standard library is, well standard, and available on all compilers so. . .)

In Python, you import modules. C++ doesnt have a module system (yet. its on the way). However, you can include other code using the #include preprocessor directive. There are actually two forms:

```
#include<>
```

and

```
#include ""
```

The difference between the two forms is actually implementation dependent technically, but all of the implementations implement the following behavior:

The *#include “”* form searches first in the same directory as the file in which the include directive exists, then searches in directories explicitly passed to the compiler by the include flag, and finally in predefined locations specific to each OS.

The *#include <>* form only searches for the target file in any include paths passed to the compiler via the include flag, and then in predefined locations specific to each OS.

As an example, the directive `#include <iostream>` imports the iostream library which makes a number of functions and classes available to us in the `std` namespace. A namespace is a label used to disambiguate symbol names in an attempt to avoid clashes. Namespaces prefix labels and are joined using ‘::’. For example `std::cout` from above references *cout* in the *std* namespace.

The main function

Every executable has an entrant function called main, which is invoked when executing the program. The Python equivalent, and there isn't a perfect one, would probably be this:

```
if __name__ == "__main__":
    from sys import argv
```

In C++, the main function has three valid forms. All the forms have one thing in common: they each must return an integer representing success or failure. Under normal circumstances, you return “0”. If there are issues, you can return an error code, which is os specific.

The first Form of main

The first form is:

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

It is a function with no arguments, which must return a *bool*, as mentioned above. This form is the easiest to write, but the least useful.

The Second Form of main

The second, and most popular form of main, takes two arguments. The first is the number arguments passed to main when the program is run. The second is a list of c string arguments, passed on the command line when the program is run.

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.

The Third Form of main

The third form adds a third parameter to main which is a c string array of environment variables:

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

Each string in the env array takes the form *VAR=Value*. We will learn later how to separate the VAR from the Value in order to make use of this information.

Namespaces

C ++ allows you to wrap code in a namespace in order to organize and protect symbols. Namespaces are roughly equivalent to Python's package definitions. However, unlike packages, which rely on the directory structure to form the shape, C++'s namespaces are explicitly defined with the *namespace* keyword.

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

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

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

Namspaces are defined thusly in C++:

```
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 preprocessor runs as the first step in compilation. We have already encountered the one of the primary roles of the preprocessor - to include other files into cpp files. However, this is far from the only use for the preprocessor. We will encounter more uses as we continue to explore C++. Some of those additional

uses are: to define macros; to conditionally define blocks of code; to avoid including the same file multiple times...