C++ for Python Programmers

Adapted from a document by Rich Enbody & Bill Punch of Michigan State University

Purpose of this document

This document is a brief introduction to C++ for Python programmers -- something to help start a transition to C++. It is not meant to be a complete introduction to C++. Brevity took precedence over details.

Why C++?

Why should Python programmers care about C++?
C++ and its ancestor language C allow one to operate closer to the operating system and hardware. That closeness can result in better performance, but it is not guaranteed. Also, C++ is a common and powerful language so it is worth learning. Look "under the hood" of many applications and operating systems, and you find C++ or C.

One thing to keep in mind when learning C++ is to keep an eye out for the STL: the C++ Standard Template Library. The STL is frequently overlooked in textbooks, but it is a powerful set of tools and will be your friend.

There are other languages worth learning, but this page is devoted solely to the transition from Python to C++. Note that C++ preceded Java and C# so one finds many similarities (as well as significant differences). Once one learns C++ it is relatively easy to transition to the other two.

Finally, language choice need not be an "either/or" situation. You can code in Python and then incorporate C++ where you need speed or low-level access to system components. It is relatively easy to wrap C++ libraries for calling from Python (many useful ones are already wrapped for you). Python plays well with other languages. Tools such as Bgen, PySTL, and Sip help.

Here at Michigan State University our first course for Computer Science and Computer Engineering majors is in Python, and the second course uses C++ in a Linux environment.

Hello World!

The tradition in C programming is to write one's first program to simply print "Hello World!" That is how the first C programming book The C Programming Language by Kernighan and Ritchie began. Since C++ is derived from C, we'll start the traditional way.

```
#include <iostream>
```

```
using namespace std;
int main()
{
	cout << "Hello World!";
	return 0;
}
```

The first thing a Python programmer notices is significant overhead to simply print a string. The #include works like `import` in Python (not really, but the analogy works for now) so in this case the input/output stream library is brought in. In this case, that is where `cout` is defined. The `using namespace` specifies the namespace so we can use `cout` instead of `std::cout`.

The `int main()` begins the declaration of the `main` function which in C++ indicates the first function to executed, i.e. "begin here." The curly braces `{` and `}` delimit the beginning and end of a block of code, in this case the beginning and end of the function `main`.

The statement `cout << "Hello World!";` directs the string to `cout`, the standard output stream, which displays it. Indentation is optional in C++, but code will be unreadable, if you do not indent. If you indent like you were forced to do in Python, your code will be readable.

Gotcha #1: Statements in C++ end with a semicolon `";"` (that isn't precisely correct, but will do for now).

Running the program.

C++ is a compiled language so you must compile it before executing it. Compilation takes the C++ code and translates it into machine language that a processor understands. Compiling and executing code is system dependent. For simplicity, I will describe how to do it in Linux.

The program must be written using some editor such as VI or EMACS on Linux. A C++ program file extension is `.cpp` so save the program as `hello.cpp`.

The GNU compiler, g++, is widely available so compiling is simply the line:
`g++ hello.cpp`

If compilation is successful, the executable code will by default be written to a file named `a.out` which can be executed by simply typing `a.out` at the command prompt. If the compiler finds syntactical errors in the program, error messages will be printed.

On a Windows computer the Visual C++ development environment is an excellent, but sophisticated, way to create and run programs (named VisualStudio). You compile and

edit within the same application. Compilation is done by "building" the program. Running is simply a command in the application.

C++ is strongly typed

C++ is a strongly typed language -- not precisely strongly typed, but close enough for now. That is, all objects (things you name) must be declared before use. Python figures out types based on the objects you assign names to, but C++ wants to know up front. Python allows names to refer to different types of objects while the program is running; in C++ the type remains fixed.

For example, the type of a variable must be stated before use.

```
int x; x = 2;
```

Gotcha #2: Forgetting to declare a variable before use.
(Don't worry the compiler will yell at you, if you forget.)
Note how each statement ends with a semicolon (Gotcha #1).

C++ types

Some C++ types are familiar:

- int
- float
- bool: same as Boolean
- string: may require #include<string>

Some are unfamiliar:

- char: single character, but a string of length one is not equivalent.

Some differences:

- `int` doesn't have extended precision. An `int` is limited by machine word size (usually 32 bits, but 64-bit use is growing); roughly, a max of 2 billion.
- There are many variations of `int`: short, long, unsigned, unsigned long, etc., but you can go far with only `int`.

Assignment

Assignment is familiar:

```
 x = 4;
```

but, multi-assignment is **not** allowed:

```
x,y = y,x
```

Variables are different.

In C++, variables are memory locations with names. Declaring a variable puts a name to that memory location and associates a type with that memory location.

```
int x, y; x = 7; y = x;
```

In Python the expression `y = x` gives two names (x and y) to the same object, but in C++ that same expression copies the value of x into the location named by y.

This difference can be illustrated as follows. In this example, we can see how two names for the same object allow changing the value using one name to be observed when displaying the other.

In Python we can name a list A and then also name it B. The 'is' operator shows that A and B name the same object. When you modify B by changing the first element in the list, that change is reflected when you display A:

```
>>> A = [2,3] >>> B = A >>> B is A True >>> B[0]=100 >>> A
[100, 3]
```

Illustrating something similar in C++ takes a bit more. In this case, we will use the C++ array which is has some similarity to a Python list except that an array's length is fixed -- it cannot change. In C++ it is not possible to assign one array to another. Instead, you have to loop through the arrays assigning one element at a time. The same is true for output.

```
#include <iostream>
using namespace std;

int main()
{
     // declare array A of length 2 and assign its elements
     // to have values 2 and 3
     int A[2] = {2,3};
     // declare array B of length 2, but don't assign its
     // elements (yet)
     int B[2];
     // loop through both arrays A and B assigning to each
     // element of B the corresponding value of A
     for (int i = 0; i < 2; i++)
     {
          B[i] = A[i];
     }
     // change the first element of B
```

```
    B[0]= 100;
    // loop through A and B printing out each element
    for (int i = 0; i < 2; i++)
    {
        cout << "A,B: " << A[i] << ", " << B[i] << endl;
    }
    return 0;
}
```

In the case of C++, the values of A are not changed when B is changed.

Arithmetic

Arithmetic expressions in C++ will be familiar:

- +, -, *, /, %
- +=, -=, *=, /=
- but no **

Increment is new:

- y = ++x;
  equivalent to x = x + 1; y = x;
- y = x++;
  equivalent to y = x; x = x + 1;

Decrement is similar: --

Relations

Relational operators in C++ wll be familiar:

- ==, !=, <, >, <=, >=

but

- && means and
- || means or

Gotcha #3: In C++ this expression does **not** work like Python or mathematics:
Assume that x = 3, what is the value of the expression: `5 < x < 10`?

In C++ the sub-expression `5 < x` is evaluated first to be false with value "0". The "0" replaces the sub-expression so next `0 < 10` is evaluated to be true. So the value of the whole expression `5 < x < 10` when `x = 3` is evaluated to be TRUE!

To get the correct answer in C++ the expression must be written as:
`(5 < x) && (x < 10)`

Conditional: if

In C++, conditionals are familiar, but

- parenthesis are required around the expression
- colon is not required
- indentation is not required, but strongly recommended.

For example:

```
if (x > 4)
       y = 7;
```

Like Python, C++ allows an "else" clause:

```
if (x <= 5)
       y = 8;
else
       z = 10;
```

However, C++ does **not** have the "elif" keyword. Instead, you must use "else if":

```
if (y == 7)
       x = 2;
else if (x != 2)
       y = 5;
else
       z = 10;
```

Python suites == C++ blocks

Python uses indentation to indicate suites.
C++ uses curly braces, `{` and `}` to delimit blocks.
Indentation is strongly recommended.

For example:

```
if ((x <= 5) && (y != 7))
{
       z = 12;
       w = 13;
}
```

Comments

Comments and their usage will be familiar, but the delimiters are different:

```
 // this is a comment at the end of a line
/* this is
       a multi-line comment */
```

C++ has no notion of a docstring.

while loop

The C++ while loop will be familiar, and shares syntax rules with the C++ "if" statement:

- parenthesis required for the expression
- curly braces delimit block (required for multi-statement)

For example:

```
while (x < 5) {   y = z + x;   x = x + 1; }
```

The C++ while also has the familar:

- continue
- break

but there is **no** "else" for a C++ while statement.

for loop

The "for" in C++ is quite different than in Python. There is good news and bad news. The bad news is that Python's powerful `"for ... in ..."` loop does **not** have an equivalent in C++.

However, the good news is that the C++ "for" loop is similar to Python's `"for x in range(...)"` loop.

For example, in Python:

```
for x in range(5):    print x,
```
the equivalent in C++ is:
```
for (int x = 0; x < 5; x++)
       cout << x;
```

The C++ "for" header has three parts:
for (int x = 0; x < 5, x++)

- the initial expression "int x = 0" which is executed once at the beginning
- the conditional expression "x < 5" which is executed before every iteration. if the condition is true, the loop body is executed
- the final expression "x++" is executed at the end of the loop body on every iteration

Note that if the conditional expression is false at the beginning, the loop body will not be executed at all.

Like "while", the "for" in C++ has

- continue
- break

but also **no** "else" in C++ "for" loop.

Equivalence of "for" and "while" in C++

In C++, "for" is a special case of "while." Understanding that relationship is important for understanding how the loops work. Note, that this equivalence is not true in Python.

To illustrate with an example:

```
for (int x = 0; x < 5, x++)
       cout << x;
```

is equivalent to

```
int x = 0;
while (x < 5)
{
       cout << x;
       x++;
}
```

Output

Output in C++ looks familiar, but with differences. In particular, Python uses syntax that is similar to C output expressions which are allowable in C++.

For example, in Python we can use:

```
print "int %d and float %5.3f" % (45, 3.1416)
```

In C (which is allowed in C++) the output looks similar:

```
printf("int %d and float %5.3f \n", 45, 3.1416);
```

Note the "\n" -- an explicit carriage return which is required in C to get the same output in this example.

However, the C++ standard is quite a bit different than the allowable C syntax.

First, C++ needs `#include <iomanip>` at the top of the program (along with `#include <iostream>`).

```
cout<< setprecision(4);
// significant digits is 4
cout<< "int "<< 45 <<" and float "<< setw(5)<< 3.1416;
cout << endl;
// endl is an explicit carriage return
```

File Input and Output

Here we see a huge difference in necessary syntax. In particular, Python's shortcut for file input is considerably shorter than the C++ syntax.

For example, in Python we could simply do:

```
for line in file("Data.txt"):
      print line
```

Alternatively, in Python we could have explictly opened the file first -- a sequence which is more similar to what is done in C++:

```
inStream = open("Data.txt", "r")
for line in inStream:
      print line
```

In C++, we must include the file stream libraries, declare objects, explicitly open the file, and explictly check for when we reach the end of the file:

```
#include <iostream>
// include library for file streams
#include <fstream>
#include <string>

using namespace std;

int main()
{
     // declare the input stream
     ifstream InStream;
```

```
      // a string
      string line;
      // open file
      InStream.open("Data.txt");
      // read a line and check for end-of-file (eof)
      while (getline(InStream, line))
      {
            // output the line
            cout << line << endl;
      }
      return 0;
}
```

Gotcha #4: Note how the "while" loop functions: we get some input (using getline) **before** we check for the end-of-file at the top of the loop and then we get the next input at the **bottom** of the loop. A common error with "while" loops is to put the "getline" at the beginning of the block in the "while" loop resulting in one too many iterations. (This gotcha also applies to Python, but in Python you likely used "while" less frequently than you will in C++.)

Functions

Functions have similarities, but have some very significant differences. **You will need to be careful!!**

Let's begin with what is similar.

In Python:

```
def fun(x):     return x + 2
```

In C++, it looks quite similar. Most notable is the need to declare the type of the parameter x.

```
int fun(int x)
{
       return x + 2;
}
```

In both C++ and Python functions return exactly one object. However, in Python you have built-in objects that contain multiple values such as tuples. While it is possible in C++ to build similar objects, the tendency is to use parameters to "return" multiple values -- similar to how mutable parameters can be changed in Python.

Parameters: value vs. reference
The default parameter passing in C++ is "pass by value" -- similar to Python passing of immutable objects. That is, changing the associated object in the function does not

change the original. Stated another way: changing the parameter does not change the argument.

Pass by value

Let's examine a C++ pass-by-value example. The output can be seen in the comments. In particular, note that changing the x in the function did not change the x in the main:

```
int fun(int x)
{
	x = x + 2;
	// prints 5
	cout << x;
}

int main()
{
	int x = 3;
	fun(x);
	// prints 3, i.e. unchanged as expected
	cout << x;
	return 0;
}
```

Pass by reference

In C++ pass-by-reference is flagged by appending an ampersand, &, to each parameter name you want passed by reference. In this example, the x in the function **refers** to the same object as the x in main so changing the value of x in the function changes the value of x in main (because they refer to the same object). It is worth noting that if the parameter in the function was named y instead of x, the behavior of the example would be unchanged.

For example, notice how in this example the value of x in main is changed by the function.

```
int fun(int& x)
{
	x = x + 2;
	// prints 5
	cout << x;
}
int main()
{
	int x = 3;
	fun(x);
	// prints 5, DIFFERENT than above!
```

```
        cout << x;
}
```

Classes

Classes have similarities, but are quite different.
More to come...

Pointers

Pointers will be new to you so we'll briefly mention them -- they require considerably more space than we can devote to the topic here. In many respects pointers are simple, but their simplicity allows one to get into trouble. A pointer is simply a variable that holds a memory address. In a variable declaration an asterisk, *, indicates that a variable is a pointer, and the type indicates the type of the object found at the address in the pointer. An ampersand, &, indicates the address of an object. For example:

```
  int A, *C = &A;
```

The pointer C (indicated by the *) holds the address (&) of integer A. Here is a simple example of a C++ program that uses pointers.

```
#include <iostream>
using namespace std;

int main()
{
   int A = 15, B = 38, *C = &A;
   cout << endl;
   cout << &A << "    " << A << endl;
   cout << &B << "    " << B << endl;
   cout << &C << "    " << C << "    " << *C << endl;
   A = 49;
   cout << endl;
   cout << &A << "    " << A << endl;
   cout << &B << "    " << B << endl;
   cout << &C << "    " << C << "    " << *C << endl;
   C = &B;
   cout << endl;
   cout << &A << "    " << A << endl;
   cout << &B << "    " << B << endl;
   cout << &C << "    " << C << "    " << *C << endl;

   return 0;
}
```

Here is sample output from a 64-bit machine. Addresses may vary from one execution to another.

```
0x7ffe6212b6cc    15
0x7ffe6212b6c8    38
0x7ffe6212b6c0    0x7ffe6212b6cc    15

0x7ffe6212b6cc    49
0x7ffe6212b6c8    38
0x7ffe6212b6c0    0x7ffe6212b6cc    49

0x7ffe6212b6cc    49
0x7ffe6212b6c8    38
0x7ffe6212b6c0    0x7ffe6212b6c8    38
```

Exceptions

Try/except are quite similar.