Transitioning from Python to C++

Based on a document by legendary section leaders Jillian Tang and Ethan Chi

Denoting Structure: Semicolons, Parentheses, and Braces

In Python, indentation and whitespace indicates where statements end and how structures nest inside one another. In C++, you need to explicitly indicate this using semicolons, parentheses, and braces.

Curly braces – the { and } characters – are almost exactly equivalent to Python indentation. You'll need to use them in if statements, for loops, while loops, and functions. For example:

Python

```
def my_function(a, b):
    if a == 1:
        print(b)
```

C++

```
void myFunction(int a, string b) {
    if (a == 1) {
        cout << b << endl;
    }
}
```

Although indentation alone does not tell the C++ compiler how structures nest inside one another, it's important to indent things nonetheless to better convey the meaning of your code.

When using curly braces, it's customary to put each close brace on its own line and aligned with the part of the code it's closing out. This makes it easier to see how things are nested.

Parentheses – the (and) characters – also have the same meaning as in Python. The difference is that, in C++, several structures require the use of parentheses in places where they were optional in Python. For example, the conditions in an if statement and while loop require parentheses:

Python

```
while x < 137:
    x = 3*x + 1
    if x % 2 == 0:
        x /= 2
```

C++

```
while (x < 137) {
    x = 3*x + 1;
    if (x % 2 == 0) {
        x /= 2;
    }
}
```

One of the more obvious differences between Python and C++ is the use of semicolons – the `;` character. In Python, the end of a statement is denoted by a newline. In C++, every statement (except for control statements like `for`, `if`, and `while`) must end with a semicolon. For example:

C++

```
int number = 137;
callAFunction(arg1, arg2);
```

However, make sure that you do ***not*** put a semicolon after a control statement like `for`, `if`, or `while`. Similarly, do not put a semicolon at the end of a statement beginning with `#`.

Bad C++: Do Not Do This!

```
#include "strlib.h";                // <-- Oops, no semicolon here!
if (myNumber == 137); {             // <-- Oops, no semicolon here!
    while (myNumber % 2 == 1); {    // <-- Oops, no semicolon here!
        myNumber /= 2;
    }
}
```

Types

C++ is a typed language, which means that you sometimes need to explicitly say what type something is.

A type is a fundamental kind of value. Examples include `int`, `string`, `char` (single character, not in Python), `double` (equivalent of Python float). You must explicitly state the type when declaring a variable, but not while using it after that. For example:

C++

```cpp
int number = 137; // Declare number; type needed
number += 106;    // number already declared; do not include type
```

Function parameters must also have types; also, every function must include a return type. If the function doesn't return anything, it has return type `void`. However, you don't have to include the types when calling the function.

Python

```python
def pizkwat(a, b):
    return a + b


def squigglebah(a, b):
    print(a + 2 * b)


ooboo = pizkwat(1, 2)
squigglebah(3, 4)
```

C++

```cpp
int pizkwat(int a, int b) {
    return a + b;
}

void squigglebah(int a, int b) {
    cout << a + 2 * b << endl;
}

int ooboo = pizkwat(1, 2);
squigglebah(3, 4);
```

For Loops

In Python, iterating over a range of numbers can be done using the `for` … **in** loop. In C++, the syntax is a bit more involved:

Python

```python
for i in range(10):
    print(i)
```

C++

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

When iterating over containers, the syntax in Python and C++ gets more similar:

Python

```python
text = # … something
for ch in text:
    print(ch)
```

C++

```cpp
string text = /* … something … */
for (char ch: text) {
    cout << ch << endl;
}
```

Conditionals

The `if` and `else` keywords work basically the same way in C++ as they do in Python:

Python

```python
if myNumber == 137:
    print("Huzzah!")
else:
    print("Alas!")
```

C++

```cpp
if (myNumber == 137) {
    cout << "Huzzah!" << endl;
} else {
    cout << "Alas!" << endl;
}
```

In C++, there is no `elif` keyword. Instead, write out `else if` as two words, like this:

Python

```python
if myNumber == 137:
    print("Yeehaw!")
elif myNumber == 106:
    print("Golly gee!")
else:
    print("Oh fiddlesticks.")
```

C++

```cpp
if (myNumber == 137) {
   cout << "Yeehaw!" << endl;
} else if (myNumber == 106) {
   cout << "Golly gee!" << endl;
} else {
   cout << "Oh fiddlesticks." << endl;
}
```

In Python, you use `and`, `not`, and `or` to combine or modify predicates. While these keywords will technically work in C++, it's not considered good style to use them. Instead, use the (slightly more cryptic, but more standard) symbols

`&&` in place of `and`   `||` in place of `or`   `!` in place of `not`

For example:

Python

```python
if a == b and b == c:
    print("Nowruz")
elif a == b or b == c:
    print("Rosh Hashanah")
elif not predicate(a):
    print("Losar")
else
    print("Just a day.")
```

C++

```cpp
if (a == b && b == c) {
    cout << "Nowruz" << endl;
} else if (a == b || b == c) {
    cout << "Rosh Hashanah" << endl;
} else if (!predicate(a)) {
    cout << "Losar" << endl;
} else {
    cout << "Just a day." << endl;
}
```

In Python, you can chain inequalities together. In C++, you cannot do this, and instead need to build multiple inequalities and use `&&` to combine them together:

Python

```python
if 0 <= a < 10:
    print("One digit")
```

C++

```cpp
if (0 <= a && a < 10) {
    cout << "One digit" << endl;
}
```

Comments

Python has single-line comments that start with #. In C++, we use // instead.

Python

```python
sporgle(quizbah) # Transform input
```

C++

```cpp
sporgle(quizbah) // Transform input
```

C++ also has multiline comments that can be used to describe a dense block of code. They begin with the sequence /* and end with */. For aesthetic reasons it's common to see each line of the comment starting with a star, but this isn't strictly necessary.

Python

```python
# Did you know that the ocean sunfish
# is so large that, when a single
# sunfish is accidentally caught by
# a fishing boat, it can account for
# about half the total catch?
```

C++

```cpp
/* Did you know that the ocean sunfish
 * is so large that, when a single
 * sunfish is accidentally caught by
 * a fishing boat, it can account for
 * about half the total catch?
 */
```

Python uses docstrings to document what a function does *inside* the body of the function. In C++, the convention is to use a multiline comment *before* the body of the function:

Python

```python
def phchthshkh(o):
    """This function name and argument
    name are terrible. They're just
    examples."""
    return o * o
```

C++

```cpp
/* This function name and argument
 * name are terrible. They're just
 * examples.
 */
int phchthshkh(double o) {
    return o * o;
}
```

Function Prototypes

C++ (for the most part) uses a model called *one-pass compilation*. This means that the C++ compiler starts at the top of the program, reading downward, and only knows about functions that it's seen so far. As a result, if you want to call a function that you will eventually define but haven't yet gotten to writing, you need to include a *prototype* for that function at the top of the program.

Python

```python


def eclair(n):
    return croissant(n)


def croissant(n):
    return n + 1



print(eclair(137))
```

C++

```cpp
int croissant(int n); // Prototype

int eclair(int n) {
    return croissant(n);
}

int croissant(int n) {
    return n + 1;
}

int main() {
    cout << eclair(137) << endl;
    return 0;
}
```

Strings and Characters

C++ makes a distinction between strings and characters. A *character* (denoted by the type `char`) is a single glyph you can display on the screen. A *string* (denoted by the type `string`) is a sequence of zero or more characters. Anything in single quotes (e.g. `'a'`) is considered a `char`, while anything in double-quotes is considered a string (e.g. `"a"`). For example:

Python

```python
papyrus = 'a'    # String of length 1

quill = "a"      # String of length 1

quill = papyrus # Sure, no problem
papyrus = quill # Sure, no problem.
```

C++

```cpp
char papyrus = 'a'; # Character

string quill = "a"; # String

quill = papyrus; # Error! Wrong types
papyrus = quill; # Error! Wrong types
```

Many string functions built into Python are not present in C++, but you can get the same functionality by using functions from the `"strlib.h"` header file.

Python

```python
text = "Pirate"
if text.startswith("Pi"):
    print("A circle")

if text.endswith("irate"):
    print("It's angry!")

if "ra" in text:
    print("Sun god!")

if text.find("at") != -1:
    print("Preposition!")

print(text.lower())
print(text.upper())

text = "a walk in the park"
parts = text.split(' ')

print(parts[0])

for part in parts:
    print(part)

text = "137"
value = int(text)
text = str(value)
```

C++

```cpp
#include "strlib.h"
#include "vector.h"

string text = "Pirate";
if (startsWith(text, "Pi")) {
    cout << "A circle!" << endl;
}

if (endsWith(text, "irate")) {
    cout << "It's angry!" << endl;
}

if (text.find("ra") != string::npos) {
    cout << "Sun god!" << endl;
}

if (text.find("at") != string::npos) {
    cout << "Preposition!" << endl;
}

cout << toLowerCase(text) << endl;
cout << toUpperCase(text) << endl;

text = "a walk in the park";
Vector<string> parts =
     stringSplit(text, " ");
cout << parts[0] << endl;

for (string part: parts) {
    print(part)
}

text = "137";
int value = stringToInt(text);
text = to_string(value);
```

Substrings

Substrings in C++ work differently than in Python. In Python, the notation `str[start:end]` gives you a substring starting at position `start` and ending at position `end`. In C++, the function call `str.substr(start, len)` gives you a substring of length `len` starting just before position `start`. You can also use `str.substr(start)` to get a substring starting at position `start` and continuing to the end of the string. Negative indices are not allowed.

Python	***C++***
`text = "appraising"`	`string text = "appraising";`
`# praising`	`// praising`
`print(text[2:])`	`cout << text.substr(2) << endl;`
`# raisin`	`// raisin`
`print(text[3:9])`	`cout << text.substr(3, 6) << endl;`
`print(text[3:-1])`	`cout << text.substr(3, text.size() - 1) << endl;`