Python to C/C++

Fall 2011

1. Main Program

Python: Program code is indented after colon ":"

```
def main():
    body of program
```

C/C++: Have more setup overhead.

Both require `#include` *directives* to access libraries of subroutines
Both require that code be enclosed within { } braces.
C++ (only) requires the `using namespace std;` *directive.*

C:

```
#include <stdio.h>      // access standard I/O routines

int main()
{
      body of program
}
```

C++

```
#include <iostream>     // access iostream library
using namespace std;    // namespace standard directive

void main()
{
    body of program
}
```

Note: Unlike in Python all statements in C/C++ terminate with a semicolon ";"

2. Variables

Python: variables are declared ***implicitly*** with the ***type*** (int, float, string) determined by the initial value assigned

```
number = 17
x = 3.14159
str = "Hello World"
```

C/C++: All variables which are typed must be ***explicitly*** declared before any code. C++ is more relaxed; variables need only be declared before they are used; initializations are optional.

```
int number = 17;
float x = 3.14159;
char str[] = "Hello World";          // C only
string str = "Hello World";          // C++
```

Variable types likes **`int`** (integer), **`char`** (character), and **`float`** (or **`double`**) are simple types; arrays and **`string`** types are structured types. In particular strings can either be implemented as type **`string`** or as arrays of characters (called `cstrings`) which terminate with a null character (`'\0'`)

This is one of the big differences between Python and C/C++. Variables in Python are *dynamically* allocated; that is as a variable is needed, memory is allocated for the variable. This is also why a variable can change its type. On the other hand in C/C++ variables are static; memory for each variable must be allocated head of time and since different data types require different amounts of memory, the type of a variable cannot change.

3. Assignment Statements

For both Python and C/C++ syntax and semantics are the same except that statements in C++ must terminate with a semi-colon ";". Unlike Python, C/C++ does not support simultaneous assignment statements.

Note Python uses the *sticky note model* for assigning values to variable; C++ uses the *post office box model*.

C/C++ has a richer set of algebraic operators including pre and post-fix increment. For example in Python to increment a variable `i`, you need the assignment statement

```
i = i+1
```

In C/C++ you can use

```
i++;  // post-fix increment
```

or

```
++i;  // pre-fix increment
```

which increments i. The two statements above produce the same results. However, you can combine pre and post fix increments within other statements . For example

```
k = i++;          // assign i to k then increment i
n = ++i;          // increment i then assign the value to n
```

4. Input/Output

Python: `input` and `print` statements; `input` functions like a simultaneous assignment statement with a built in prompt. If multiple values are read, they must be separated by commas.

```
a,b = input("Enter two integers ")

print x, y
```

Note that print always advances to the next line; to not advance to the next line, end with a comma

```
print x, y,
```

In Python formatted output is done using the "`%`" string formatting operator. The `%` symbols does double duty as it's also used with the template-string as a field specifier where `%d, %f`, and `%s` specifiers denote fields for integer, floating point and string variables.

```
print "The answer is %d\n" % (ans)
```

Python also has `raw_input("  ")` which returns any input as a string of characters

C uses the functions `printf()` and `scanf()`, `putchar()` and `getchar()` to write and read formatted output/input of single character output/input.

```
printf("%s", "Enter two integers ");
scanf("%d %d", &a, &b);
```

The syntax of both `printf()` and `scanf()` require that the first parameter be a format string (this is similar to formatting output in Python). Fields for `int` (integer), `float`, `char` (character) or `string` variables are designed by `%d`, `%c`, `%f`, and `%s`. There must be agreement in number and type between the variables and/or expressions that follow the format string and the format fields.

Optionally format fields `%`*n*`d`, `%`*n*`c`, `%`*n*`.`*m*`f`, and `%`*n*`s` where *n* and *m* are integers can be used to allocate *n* columns and in the case of `float` variables *m* digits to the right of the decimal point for the corresponding variable.

With `scanf()` the variables the follow the format string must be prefixed with '`&`'since pass by reference variables are used (see section on parameter passing). There is an exception to the use of the '`&`"; if the parameter is an array, the `&` is not used (the reason has to do with the way arrays are implemented).

If multiple variable re read, they only need be separated by whitespace (i.e. blanks, tabs, or newline characters (`'\n'`); Python uses commas as separators.

For single character input, `getchar()` returns the next character input by the user. For example

```
ch = getchar();
```

For character output, `putchar(ch)` outputs the character `ch`.

Note: Most of the assemble languages we'll be studying in Comp 255 permit only single character I/O like C's `getchar()` and `putchar()`.

C++ uses `cin >>` and `cout <<` to read and write.

```
cout << "Enter two integers ";
cin >> a >> b;
```

Note that any prompt must be done with cout <<. If multiple values are read in, unlike Python they must be separated by whitespace characters; that is blanks, tabs or newline characters. A comma is not whitespace.

```
cout << x << y;
```

Unlike Python, C++ does not automatically advance to the next line. To do so you must explicitly output a newline character, denoted by '\n'

```
cout << x << y << '\n';
```

Alternately you can use `endl`

```
cout << x << y << endl;
```

Note: The *syntax* of I/O for C++ is derived from the *semantics* of I/O. In C++ I/O is seen as a stream of text characters either directed to the console output device (abbreviated `cout`) or taken from the console input device (abbreviated `cin`). The << and >> are called insertion and extraction operators. So `cout << number;` means

insert the value of number into the output stream to the console output device and `cin >> number;` means extract from the input stream the value for number and store in number. This is why we say `cout << x << y;` and `cin >> a >> b;` using the << (insertion) and >> (extraction) operators between values/variables instead of using commas.

Hello World Examples:

Python

```
def main():

    print("\nHello World\n")
```

C

```
#include <stdio.h>      // include I/O library

int main()
{
    printf("\nHello World\n");
}
```

C++

```
#include <iostream>     // include I/O stream library
using namespace std;

void main()
{
    cout << "\nHello World\n";
}
```

5 if else statements

Python:

```
if number == 0:
    <if true clause>
else:
    <else false clause>
```

Note the bodies of the <if true clause> and <else false clause> are indented

C/C++

```
if (number == 0)
    {
      <if true clause>
    }
else
    {
      <else false clause>
    }
```

Note the bodies of the <if true clause> and the <else false clause> are enclosed between within { } braces. Also the condition being tested, (number == 0), must be enclosed by parentheses.

Multi-way branching: Python uses the `elif` statement (a contraction of else if); C++ uses `else if`.

6. Counting Loops

Python for loop

```
for i in range(10):
    <body of loop>

for i in [2, 3, 5, 7, 11]:
    <body of loop>
```

In Python the loop index i *iterates over the sequence* taking on in turn each value in the sequence and executing the body of the loop. The general form of the Python for loop is

```
for <var> in <sequence>:
    <body of loop>
```

C/C++ for loop

```
for (int i = 0; i < 10; i++)
    {
    <body of loop>
    }
```

In C++ the loop index i is initialized (`i = 0`), tested (`i < 10`) and if the test is true the body of the loop is executed and then the loop index is incremented (`i++` or `i = i + 1`). This is equivalent to the first Python for loop above. Unfortunately the second Python for loop cannot be implemented in C++ using a for loop. On the other hand there are some things a C++ form loop can do that a Python for can't. The general form is

```
for (initial step; terminal condition; update action)
    {
    <body of loop>
    }
```

7. While Loops

Like if else statements while loops in Python and C++ are similar in syntax and semantics, the main difference being that Python uses a colon and indents the body of the loop while C++ requires parentheses around the test condition and encloses the body of the loop within { } braces.

All three languages support a `break` statement to exit a loop immediately

Python while loop

```
while a % b != 0:
    b = b+ 1
```

C/C++ while loop

```
while (a % b != 0)
    {
    b = b+1;
    }
```

Unlike Python C/C++ has an explicit bottom test loop: the `do while` loop. The syntax and semantics should be obvious. Note that a bottom test loop always executes at least once.

```
do
    {
    b = b + 1;
    } while (a % b != 0)
```

Note: Python would implement the above bottom test loop using a break statement. Note the reversed test logic used by the Python version.

```
while true:
    b = b + 1
    if a % b == 0:
        break;
```

8. Grouping Statements – indenting vs {}

Python uses indentation to group statements that are subordinate to an `if, elif,` or `else` statement that make up the body of a loop. C++ uses pairs of braces { } to do the same. In C++ a set of statements enclosed by { } is called a compound statement.

Euclidean Algorithm Examples

Python – Note the efficient use of Python's simultaneous assignment statements.

```
def main():

    a, b = input("\nEnter two integers separated by a comma: ")
    while a % b != 0:
        a, b = b, a % b
    print "\nThe gcd is %d\n" % (b)
```

C – Since C does not support simultaneous assignment statements, a third variable r is needed to hold the remainder a % b.

```
#include <stdio.h>

int main()
{
	int a, b, r;
	printf("\nEnter two integers: ");
	scanf("%d %d", &a, &b);	// note pass by reference
	while (a % b != 0)
	{
		r = a % b;
		a = b;
		b = r;
	}
	printf("\nThe gcd is %d\n", b);

}
```

C++: I/O is the main difference between the C and C++ versions of the program

```
#include <iostream>

using namespace std;

void main()
{
	int a, b, r;
	cout << "\nEnter two integers: ";
	cin >> a >> b;
	while (a % b != 0)
	{
		r = a % b;
		a = b;
		b = r;
	}
	cout << "\nThe gcd is " << b << "\n";
}
```

9. Strings and cstrings

There are two string data types in C/C++. The `string` data type in C/C++ is similar in behavior to Python's string type. The more primitive `cstring` type is an array of characters (char).

For example, the declaration and initialization

```
char str0[] = "Hello";
```

creates a character array with 6 components holding the characters `'H', 'e', 'l', 'l', 'o'` and `'\0'`, the last being a null character (ASCII 0) . All `cstrings` must end with a null character.

Unfortunately `cstrings` being array must be treated like arrays. `Cstrings` cannot be assigned to `cstrings`. So for example., given the declarations

```
char str0[] = "Hello";
char str1[10];          // allocate an array of capacity 10
```

the following assignment statement will not work

```
str1 = str0;
```

Instead you must assign each character in `str0` to `str1`. This requires using a loop and knowing how many characters are in `str0.` Fortunately C/C++ provides a built-in function called `strlen()` which returns the length of the string

```
for (i=0; i < strlen(str0); i++)
{
    str1[i] = str0[i];
}
str1[i] = '\0';               // don't forget terminating null
```

A slicker way which doesn't use `strlen()` but looks for the null character `'\0'` instead is done by

```
for (i=0; str0[i] != '\0'; i++)
{
    str1[i] = str0[i];
}
str1[i] = '\0';               // don't forget terminating null
```

String to Integer Conversion Examples

Python: This program reads two strings of integers (using `raw_input()`) then converts both string to integer values. To convert a string of digits to an integer, each character/digit must be converted to its corresponding integer value. This is done by taking the ASCII code of each digit (the `ord()` function returns the ASCII code of a character) and subtracting 48. This is because the ASCII codes for the digit '0' thru '9' are 48 thru 57. Subtracting 48 gives you the integer values 0 thru 9. Before the value of each digit is added to the number, the number is multiplied by 10 so that each digit as it's read has the proper weight.

```
def main():

    str0 = raw_input("Enter an integer: ")
    k = 0
    for i in range(len(str0)):
        k = 10*k + (ord(str0[i]) - 48)

    str1 = raw_input("Enter another integer: ")
    n = 0
    for i in range(len(str1)):
        n = 10*n + (ord(str1[i])- 48)

    print "\nThe sum is %d\n" % (k+n)
```

C: Here is the same code in C. Note that we do not prefix an `&` to `str0` or `str1` in the scanf() function call since both are arrays. Also C does not need an explicit "`ord()`" function (like in Python) to return the ASCII code for a character. When C sees a character variable embedded in an arithmetic expression, it treats the character like an integer using its ASCII code value.

```
#include <stdio.h>

int main()
{
      char str0[10], str1[10];
      int i, k, n;

      printf("\nEnter an integer: ");
      scanf("%s", str0);
      k = 0;
      for (i = 0; i < strlen(str0); i++)
      {
            k = k * 10 + (str0[i] - 48);
      }

      printf("\nEnter another integer: ");
      scanf("%s", str1);
      n = 0;
      for (i = 0; i < strlen(str1); i++)
      {
            n = n * 10 + (str1[i] - 48);
      }

      printf("\nThe sum is %d\n", k+n);
}
```

10. Functions and Procedures

In Python and C/C++ functions are defined and called in much that same way.

In Python the following code defines a function which computes n factorial.

```
def factorial(n):
    p = 1
    for k in range(1,n+1):
        p = p * k
    return p
```

Functions are define before the main() program

In C/C++ the same function can be implemented by the code below. The main difference is that all parameters are must be explicitly typed in the function definition.

```
int factorial(int n)
{
    int p = 1;
    int k;
    for (k = 1; k <=n; k++)
    {
        p = p * k;
    }
    return p;
```

Examples: Using Functions to Convert String to Integers

Python: This is essentially the same program as above except the routine to convert a string of digits into an integer has been consolidated into a function

```
def str2int(s):
    n = 0
    for i in range(len(s)):
        n = 10*n + (ord(s[i])-48)
    return n

def main():

    str0 = raw_input("Enter an integer: ")
    k = str2int(str0)

    str1 = raw_input("Enter another integer: ")
    n = str2int(str1)

    print "\nThe sum is %d\n" % (k+n)
```

C: Again the same as the above with the code to convert a string of digits into an integer consolidated into a function

```
#include <stdio.h>

int str2int(char s[])
{
      int i;
      int n = 0;
      for (i = 0; i < strlen(s); i++)
      {
            n = n * 10 + (s[i] - 48);
      }
      return n;
}

int main()
{
      char str0[10], str1[10];
      int i, k, n;
      char ch;

      printf("\nEnter an integer: ");
      scanf("%s", str0);
      k = str2int(str0);

      printf("\nEnter another integer: ");
      scanf("%s", str1);
      n = str2int(str1);

      printf("\nThe sum is %d\n", k+n);
}
```

11. Parameters

In Python and C/C++ all parameters are **pass by value**; that is the value of the actual parameter (the parameter used when the function is called or invoked) is assigned to the formal parameter (the parameter used in the function definition). Thereafter the connection between the actual parameter and the formal parameter is broken so any change made to the formal parameter does not change the actual parameter. It's a one way transfer of information.

In C/C++ the number and type of the actual parameters must agree with the number and type of the formal parameters; order is important. The same is more or less true in Python but not strongly enforced.

Functions return values, hence the return statement. In C/C++ the return type must be explicitly defined in the header of the function (in the factorial example above, the initial `int` in the function header declares the return type).

In Python it is possible for a function to return more than one value; This is not possible in C/C++; functions in C/C++ may only return one value.

There are programming situations where we would like a function to either *modify* its parameters or return multiple values. Since Python allows a function to return multiple values, there are easy "work-arounds" for both of these but not so with C/C++. For example, suppose we want to write a function that exchanges two values; that is, if you passed it two variables `a` and `b` it would exchange the values so that `a` would be equal to the old value of `b` and `b` would be equal to the old value of `a`. The **pass by value** parameter passing convention does not allow changes to parameters. So how can this be done?

We'll examine first how this is done in C++ (C uses a more explicit mechanism). C++ allows a second type of parameter passing called **pass by reference**. In **pass by reference** the actual parameters and the formal parameters are **linked** during the functions call so any change to one results in a change of the other. C++ denotes **pass by reference** parameters using an "**&**" in the definition of the function (see below). So when `exchange(a, b)` is called in the `main()` function, the actual parameter `a` is linked to the formal parameter `a` (even thought they are the same name, their full names are `a.main` and `a.exchange`) and the actual parameter `b` (or `b.main`) is linked to the formal parameter `b` (or `b.exchange`). Thereafter any change to one results in a change to the other.

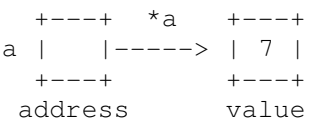
```
#include <iostream>
using namespace std;

void exchange(int &a, int &b)  // pass by reference parameters
{
      int c;
      c = a;
      a = b;
      b = c;
}

void main()
{
      int a, b;
      cout << "\nEnter two integers: ";
      cin >> a >> b;
      exchange(a, b);
      cout << "\nReversed: " << a << " " << b << endl;
}
```

C has a similar pass by reference type mechanism but it's really pass by value with a twist that makes it behave like pass by reference. In C (and in C++ but the later doesn't figure into our discussion here) you can either define a variable or define a pointer to a variable, the latter holding the address of a variable. Syntactically an asterisk is used to denote a pointer variable.

For example

```
      int c;      // define a integer variable
      int *a;     // define a pointer to an integer variable
```

Understand that `a` is the variable (not `*a`); in a variable declaration the * indicates that `a` stores the address of a variable, not the value of the variable (see diagram below). If you want the value of the variable you have to "follow the pointer" so in an assignment statement you use the expression `*a` (think of its meaning as "the value that a points to") .* is called the *de-referencing* operator. The exact meaning and use of the *, to define a pointer or de-reference a variable, depends on context.



So to assign a value to what a points to, use

```
      *a = 7;
```

Ok how does this allow actual and formal parameters to be linked so that a change in one causes a change in the other?

When we call `exchange(&a, &b)` (see below) the ampersand in the parameter calls next to the actual parameter says pass the *address* (not the value) of actual parameters a and b.

In the definition of exchange the formal parameters `a.exchange` and `b.exchange` are pointer type variables; they now hold the addresses of `a.main` and `b.main`.

The assignment statement `c = *a` assigns the what `a.exchange` points to (*a means "the value that a points to"), in this case `a.main`, to c. The next assignment statement `*a = *b` says assign what `b` points to (in this case the value of `b.main`) to what `a` points to (in this case `a.main`). So `a.main` is assigned the value `b.main`! Finally `*b = c` says assign `c` (which is `a.main`) to what `b` points to (`b.main`)

The mechanism is still **pass by value** (the addresses in `a.exchange` and `b.exchange` never change) but the effect is **pass by reference** in that the values of `a.main` and `b.main` and what `a.exchange` and `b.exchange` point to change.

```
#include <stdio.h>

void exchange(int *a, int *b)  // pointer type parameters
{
      int c;
      c = *a;           // assign to c what a points to
      *a = *b;
      *b = c;
}

int main()
{
      int a, b;                           // define two integer variables
      printf("\nEnter two integers: ");
      scanf("%d%d", &a, &b);
      exchange(&a, &b);                   // pass their addresses to exchange
      printf("\nReversed: %d %d\n", a, b);
}
```

Syntactically just remember to declare all "**pass by reference**" formal parameters using *'s, to use *'s when referring to the formal parameters in the function and to use ampersands when passing the actual parameters – and it all works out nicely.

Note final note. In C/C++ array parameters use a pass by reference mechanism but without the syntactical "`*`" and "`&`" baggage (recall the `int str2int(char s[])` function above). That's because in C/C++ arrays are actually implemented using a pointer type mechanism; de-referencing is automatic so no explicit de-referencing on the part of the programmer is need.

Rev 08/24/2011