CSCI 2100: Lecture 2
on C++

Data Types
Control Structures

Recap

- Still no dongle!
  Should be in by next week - until then, may need to take notes.

- Lab tomorrow:
  - Prelab due via email before 2pm
  - Lab available on ZyLabs by morning
- Reading on ZyLabs: by Friday
- HW1: on ZyLabs after lab tomorrow.
  due next Friday

C++ versus Python

Python

```python
def gcd(u, v):
    # we will use Euclid's algorithm
    # for computing the GCD
    while v != 0:
        r = u % v     # compute remainder
        u = v
        v = r
    return u

if __name__ == '__main__':
    a = int(raw_input('First value: '))
    b = int(raw_input('Second value: '))
    print 'gcd:', gcd(a,b)
```

C++

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

int gcd(int u, int v) {
    /* We will use Euclid's algorithm
       for computing the GCD */
    int r;
    while (v != 0) {
        r = u % v;    // compute remainder
        u = v;
        v = r;
    }
    return u;
}

int main( ) {
    int a, b;
    cout << "First value: ";
    cin >> a;
    cout << "Second value: ";
    cin >> b;
    cout << "gcd: " << gcd(a,b) << endl;
    return 0;
}
```

Figure 1: Programs for computing a greatest common divisor, as written in Python and C++.

Primitive data types

C++ Type	Description	Literals	Python analog
bool	logical value	**true** **false**	**bool**
short	integer (often 16 bits)		
int	integer (often 32 bits)	39	
long	integer (often 32 or 64 bits)	39L	**int**
——	integer (arbitrary-precision)		**long**
float	floating-point (often 32 bits)	3.14f	
double	floating-point (often 64 bits)	3.14	**float**
char	single character	`'a'`	
string[a]	character sequence	`"Hello"`	**str**

Figure 2: The most common primitive data types in C++.

[a]Not technically a built-in type; included from within standard libraries.

Ex:

```
int x = 5;
x = x + 10;
x++;
```

Operations:

	Python	C++	Description
	Arithmetic Operators		
	−a	−a	(unary) negation
	a + b	a + b	addition
	a − b	a − b	subtraction
	a ∗ b	a ∗ b	multiplication
▷	a ∗∗ b		exponentiation
	a / b	a / b	standard division (depends on type)
▷	a // b		integer division
	a % b	a % b	modulus (remainder)
▷		++a	pre-increment operator
▷		a++	post-increment operator
▷		−−a	pre-decrement operator
▷		a−−	post-decrement operator
	Boolean Operators		
▷	**and**	&&	logical and
▷	**or**	||	logical or
▷	**not**	!	logical negation
▷	a **if** cond **else** b	cond ? a : b	conditional expression
	Comparison Operators		
	a < b	a < b	less than
	a <= b	a <= b	less than or equal to
	a > b	a > b	greater than
	a >= b	a >= b	greater than or equal to
	a == b	a == b	equal
▷	a < b < c	a < b && b < c	chained comparison
	Bitwise Operators		
	˜a	˜a	bitwise complement
	a & b	a & b	bitwise and
	a | b	a | b	bitwise or
	a ˆ b	a ˆ b	bitwise XOR
	a << b	a << b	bitwise left shift
	a >> b	a >> b	bitwise right shift

Figure 5: Python and C++ operators, with differences noted by ▷ symbol.

More:

- Ints can also be unsigned:
  range from 0 to $2^b - 1$
  instead of $-(2^{b-1})$ to $(2^{b-1} - 1)$

- Strings and chars are very different:
  - Chars are actually just ASCII numbers
  - Strings - a completely different beast, & not built in

Ex

```
import <string>
using namespace std;

char a;
a = 'a';
a = 'h';

string word;
word = "CSCI 3100";
```

For more: Cplusplus.com
+ search "string"

From transition guide:

Syntax	Semantics
s.size( ) s.length( )	Either form returns the number of characters in string s.
s.empty( )	Returns **true** if s is an empty string, **false** otherwise.
s[index]	Returns the character of string s at the given index (unpredictable when index is out of range).
s.at(index)	Returns the character of string s at the given index (throws exception when index is out of range).
s == t	Returns **true** if strings s and t have same contents, **false** otherwise.
s < t	Returns **true** if s is lexicographical less than t, **false** otherwise.
s.compare(t)	Returns a negative value if string s is lexicographical less than string t, zero if equal, and a positive value if s is greater than t.
s.find(pattern) s.find(pattern, pos)	Returns the least index (greater than or equal to index pos, if given), at which pattern begins; returns **string**::npos if not found.
s.rfind(pattern) s.rfind(pattern, pos)	Returns the greatest index (less than or equal to index pos, if given) at which pattern begins; returns **string**::npos if not found.
s.find_first_of(charset) s.find_first_of(charset, pos)	Returns the least index (greater than or equal to index pos, if given) at which a character of the indicated string charset is found; returns **string**::npos if not found.
s.find_last_of(charset) s.find_last_of(charset, pos)	Returns the greatest index (less than or equal to index pos, if given) at which a character of the indicated string charset is found; returns **string**::npos if not found.
s + t	Returns a concatenation of strings s and t.
s.substr(start)	Returns the substring from index start through the end.
s.substr(start, num)	Returns the substring from index start, continuing num characters.
s.c_str( )	Returns a C-style character array representing the same sequence of characters as s.

Figure 3: Nonmutating behaviors supported by the **string** class in C++.

Syntax	Semantics
s[index] = newChar	Mutates string s by changing the character at the given index to the new character (unpredictable when index is out of range).
s.append(t)	Mutates string s by appending the characters of string t.
s += t	Same as s.append(t).
s.insert(index, t)	Inserts copy of string t into string s starting at the given index.
s.insert(index, num, c)	Inserts num copies of character c into string s starting at the given index.
s.erase(start)	Removes all characters from index start to the end.
s.erase(start, num)	Removes num characters, starting at given index.
s.replace(index, num, t)	Replace num characters of current string, starting at given index, with the first num characters of t.

Figure 4: Mutating behaviors supported by the **string** class in C++.

Mutable vs. immutable

Sanity check (& recap of last class): difference?

In C++:

Maximum flexibility - everything is mutable by default!

Even ints:

```
int x = 64;
x << 2;
```

Creating variables:

All must be explicitly created!
(& given a type)
↳ this is what "static" means

Ex:

```
int number;
int a, b;
int age(24);
int age2(currYear - birthYear);

String greeting("Hello");
```

Immutable

- You can force immutability:

```
const float gravity(-9.8);
```

Why?

Converting: be careful!

```
int a(5);
double b;
b = a;

int a;
double b(2.67);
a = b;

char x = 'a';
a = x;
```

- Can't convert string ↔ #s.
  But can convert chars ↔ #s.
  So ok: if word is a string

```
word[2]++;
word[0] = 121;
```

Arrays

- Python has lists, tuples, etc.
- C++ : starts with only arrays
  - size is fixed at time of declaration
  - type is fixed (+homogeneous)

Ex:

```
int numbers[5];
numbers[0] = 55;
numbers[3] = 20;
cout << number[0] << endl;
```

Picture

Caution:

- Seg faults will be a problem!

Ex:

```
int numbers[5];
numbers[0] = 55;
numbers[3] = 20;
numbers[5] = 5;
```

Creating arrays

```
int daysInMonth = {31, 28, 31, 30,
                   31, 30, 31, 31, 30,
                   31, 30, 31};
```

Error:

```
int daysinmonth[];
```

One exception:

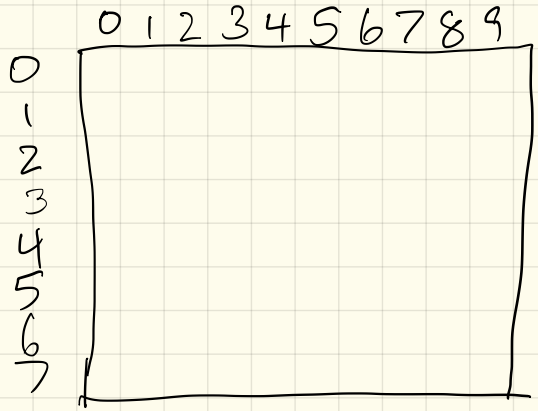
```
char greeting[] = "Hello";
```

Reason:

Multi-d arrays

```
int table[8][10];
```

Mental Picture:



"Real" picture:

Usage:

Control Structures

C++ has loops, conditionals, functions, & classes.

Syntax will be similar, but not the same.

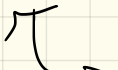
(Remember: Check Cplusplus.com or transition guide in a pinch!)

I'll do a quick overview in class

(but expect you to check syntax on these 1st questions.)

While loops

```
while (bool)
{
} body;
```



while(bool) {body;}

- bool is any boolean expression
- don't need {} if only 1 line:

```
while (a<b)
    a++;
```

- Careful! Problem?

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

For loops

Not iterator based!

Ex:

```
for (int count = 10; count > 0; count--)
    cout << count << endl;
cout << "Blastoff!" << endl;
```

Notes:

Functions Syntax:

```
void countdown( ) {
  for (int count = 10; count > 0; count--)
    cout << count << endl;
}
```

Or:

```
void countdown(int start=10, int end=1) {
  for (int count = start; count >= end; count--)
    cout << count << endl;
}
```

Conditional

```
if (bool) {
    body1;
}
else {
    body 2;
}
```

Ex:

```
if (x < 0)
  x = −x;
```

```
if (groceries.length( ) > 15)
  cout << "Go to the grocery store" << endl;
else if (groceries.contains("milk"))
  cout << "Go to the convenience store" << endl;
```

These can get a bit ugly!

```
if (cond 1)
if (cond 2)
    {code;}
else
    {code;}
if (cond 3)
if (cond 4)
if (cond 5)
    {code;}
else
    {code;}
```

Booleans & whiles/conditionals:

- If & while can both be written with numeric values as the boolean

Reason: bools are really just integers!

Ex:
```
if (mistakeCount)
    cout << "error!" << endl;
```

0 <=> false

all else is true

An error that crops up w/ conditionals / booleans:

"Feature" 1: bools are really ints, & 0 is false

"Feature" 2: operator = chains

```
x = y = 5;
```

So - a common bug:

```
double gpa;
cout << "Enter your gpa: ";
cin >> gpa;
if (gpa = 4.0)
   cout << "Wow!" << endl;
```

Do-While loops

- A variant of whiles that executes body before checking condition

```
int number;
do {
  cout << "Enter a number from 1 to 10: ";
  cin >> number;
} while (number < 1 || number > 10);
```

Main function

- Every program starts running at its main function.

Syntax

```
int main () {
    body;
    return 0;
}
```

Other functions:

```
int myAdd( int x, int y) {
    return (x+y);
}
```

Next time:

- Lab
- Friday: Sub will cover I/O
- Next week:
  no class Monday
  Then on to classes by end of the week!