Announcements

- Lab Friday, normal lecture tomorrow

- Overview of lab + software

passwd
Comparison

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++

#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;
}
White space — returns, tabs, etc. are ignored in C++

```c
int gcd(int u, int v) { int r; while (v != 0) { r = u % v; u = v; v = r; } return u; }
```

 NEVER submit this
(Recall that these were very important in python)

Here, we use ( ) and { } to mark loops, booleans, etc.
Compiling

- In Python, you save code as `gcd.py`, then type `python gcd.py` to run it.

- In C++:
  - Save as `gcd.cpp`
  - Type `g++ -o gcd gcd.cpp`
  - Type `./gcd` to compile and output
  - Or: `g++ file.cpp > ./a.out`
## Data Types

<table>
<thead>
<tr>
<th>C++ Type</th>
<th>Description</th>
<th>Literals</th>
<th>Python analog</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool</code></td>
<td>logical value</td>
<td><code>true</code> <code>false</code></td>
<td><code>bool</code></td>
</tr>
<tr>
<td><code>short</code></td>
<td>integer (often 16 bits)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>int</code></td>
<td>integer (often 32 bits)</td>
<td>39</td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>long</code></td>
<td>integer (often 32 or 64 bits)</td>
<td>39L</td>
<td><code>int</code></td>
</tr>
<tr>
<td></td>
<td>integer (arbitrary-precision)</td>
<td></td>
<td><code>long</code></td>
</tr>
<tr>
<td><code>float</code></td>
<td>floating-point (often 32 bits)</td>
<td>3.14f</td>
<td><code>float</code></td>
</tr>
<tr>
<td><code>double</code></td>
<td>floating-point (often 64 bits)</td>
<td>3.14</td>
<td><code>float</code></td>
</tr>
<tr>
<td><code>char</code></td>
<td>single character</td>
<td><code>a</code></td>
<td><code>str</code></td>
</tr>
<tr>
<td><code>string</code></td>
<td>character sequence</td>
<td>&quot;Hello&quot;</td>
<td><code>str</code></td>
</tr>
</tbody>
</table>

* `string` is not a default type in C++.
Data Types (cont)

- Ints can also be unsigned:
  instead of ranging from \(-2^{b-1}\) to \((2^{b-1}-1)\)
  go from 0 to \(2^{b-1}\).

- Strings and chars are very different.
Char versus string

```cpp
#include <string>
char a;
a = 'a';
a = 'h'; // use single quote!
```

```cpp
string word;
word = "CS 180";
```

Strings are not automatically included. Standard in most libraries, but need to import.
Strings

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

**DFM:** mutable

- changable : list

**DFM:** immutable

- not changable : int, string, tuple

Default in C++
C++: Maximum flexibility
Everything is mutable by default!

```cpp
string word;
word = "Hello";
word[0] = 'J';

word is "Jello"
```
Creating variables.

All variables must be explicitly created and given a type.

```java
int number;
int a, b;  // not int a, char b;
int age (35);
int age2 (currYear - birthYear);
int age3 (21), zipcode (63116);
String greeting ("Hello");
```
Immutable variables

We can force some variables to be immutable—use `const`:

```
const float gravity = -9.8;
```

Why? Compiler enforces no changes.
Converting between types

Be careful!

```c
int a(5);
double b;
b = a;  // b = 5.0
```

```c
int a;
double b (2.67);
a = b;  // a = 2 (8 bit ASCII)
```

```c
char x = 'a';
a = x;
x = 100- something
```
Converting with strings

- Can’t go between strings & numeric types at all.
  \[ x = "67" \]

- But chars will convert to numbers.

ASCII
Control Structures

C++ has loops, conditionals, functions, and objects.

Syntax is similar, but just different enough to get into trouble.

(Remember to use your book's index in a pinch!)  
Cplusplus.com or transition guide
While loops

\[
\text{while (bool) } \\
\text{ & body:} \\
\]

Notes:
- bool is any boolean expression
- don't need \& if only 1 command in the loop: while (a < b)
  \[
  a++ \\
  a = a + 1 \text{ or } a += 1
  \]
# Booleans

<table>
<thead>
<tr>
<th>Boolean Operators</th>
<th>Python</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>and</code></td>
<td><code>&amp;&amp;</code></td>
<td>logical and</td>
</tr>
<tr>
<td><code>or</code></td>
<td>`</td>
<td></td>
</tr>
<tr>
<td><code>not</code></td>
<td><code>!</code></td>
<td>logical negation</td>
</tr>
<tr>
<td><code>a if cond else b</code></td>
<td><code>cond ? a : b</code></td>
<td>conditional expression</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Comparison Operators</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a &lt; b</code></td>
<td><code>a &lt; b</code></td>
<td>less than</td>
</tr>
<tr>
<td><code>a &lt;= b</code></td>
<td><code>a &lt;= b</code></td>
<td>less than or equal to</td>
</tr>
<tr>
<td><code>a &gt; b</code></td>
<td><code>a &gt; b</code></td>
<td>greater than</td>
</tr>
<tr>
<td><code>a &gt;= b</code></td>
<td><code>a &gt;= b</code></td>
<td>greater than or equal to</td>
</tr>
<tr>
<td><code>a == b</code></td>
<td><code>a == b</code></td>
<td>equal</td>
</tr>
<tr>
<td><code>a &lt; b &lt; c</code></td>
<td><code>a &lt; b &amp;&amp; b &lt; c</code></td>
<td>chained comparison</td>
</tr>
</tbody>
</table>

*Error in C++*
For loops

Example:

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

Note: int declaration isn't required (as long as variable already was declared!)

```
for (int i = 0; i < bound; i++) {
```

Defining a function: example

Remember countdown function from 150?

type (no return type)

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

End function

int main() {
    countdown();
}

3
Optional arguments

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

countdown();
countdown(20, 10);
If statements

if (bool1) 3
  body1;
else 3
  body2;

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;

Note: - Don't need brackets if 1 line
  - don't need else
  - no elif
Booleans & if/whiles

If & while statements can be written with numeric conditions (which are really booleans).

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

0 => false
anything else is true