CS 2100 - Intro to C++

- HW 1 is posted

- Lab is due today

- Email me to set up office hours if needed this week
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(input('First value: '))
    b = int(input('Second value: '))
    print('gcd:', gcd(a, b))

#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++

```cpp
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` and then type "python gcd.py" to run it.

- In C++:
  * Save as `gcd.cpp`
  * Type "g++ -o gcd gcd.cpp"
  * Type "./gcd"

Optional: specify executable `gcd` program in this directory.
Other ways to compile:

\[
\begin{align*}
&\text{x++ gcd.cpp} \\
&\text{\quad b) save executable as a.out} \\
&\text{x/a.out}
\end{align*}
\]

\[
\begin{align*}
&\text{make gcd} \\
&\text{b) using a makefile}
\end{align*}
\]
## Data Types

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

\(^a\) string literals in C++ are limited to 8K characters.
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.
  - Strings must be imported
  - Chars are ' ', strings ' '
Char versus String

```cpp
import <string>
using namespace std;

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

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

Strings are not automatically included. Standard in most libraries, but need to import.
<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 (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 (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, zero if equal, and a positive value if s is greater than t.</td>
</tr>
<tr>
<td>s.find(pattern)</td>
<td>Returns the least index (greater than or equal to index pos, if given) at which pattern begins; returns string::npos if not found.</td>
</tr>
<tr>
<td>s.find(pattern, pos)</td>
<td></td>
</tr>
<tr>
<td>s.find_last(pattern)</td>
<td>Returns the greatest index (less than or equal to index pos, if given) at which pattern begins; returns string::npos if not found.</td>
</tr>
<tr>
<td>s.find_last(pattern, pos)</td>
<td></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 string::npos if not found.</td>
</tr>
<tr>
<td>s.find_last_of(charset)</td>
<td>returned 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.</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.c_str()</td>
<td>Returns a C-style character array representing the same sequence of characters as s.</td>
</tr>
</tbody>
</table>
Mutable versus immutable

**Def.:** mutable
change the value: lists

**Def.:** immutable
opposite: value is fixed: tuples or string s
C++: Maximum flexibility

Everything is mutable by default!

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

© word is now "Hello"

Even ints:
Can move bits of an int over
\( x \ll 2 \)
Creating variables

All variables must be explicitly created and given a type.

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

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

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

Why?

Enforced by compiler!

gravity = 12; ← compile error
Converting between types

Be careful!

\[
\begin{align*}
\text{int } a(5) &; \\
\text{double } b &; \\
b &= a;
\end{align*}
\]

\[
\begin{align*}
\text{int } a &; \\
\text{double } b (2.67) &; \\
a &= b;
\end{align*}
\]

\[
\begin{align*}
\text{char } x &= \text{'a'}; \\
a &= x;
\end{align*}
\]
Converting with strings

- Can't go between strings and numeric types at all.
  \[
  \text{word} = 125 \text{; } \leq \text{ NO}
  \]
  \[
  \text{char } x = 125 \text{; } \leq \text{ OK}
  \]
- But chars will convert to numbers.

\[
\text{OK:}
\]
\[
\text{word [2]} + \text{;} +
\]
\[
\text{word [0]} = 105 \text{;}
\]

ASCII
Control Structures

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

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

(Remember to use cplusplus.com or transition guide in a pinch!)
**While loops**

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

**Notes:**

- `bool` is any boolean expression
- don't need `;}` if only 1 command in the loop: `while (a < b) ++a;`

**Note:** $a++$ is

```c
a = a + 1
```

**Careful:**

```c
while (x < 0) {
    x = x + 5;
    k = cont << k;
}
```
<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>Python</th>
<th>C++</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>
For loops

Example:

```cpp
for (int count = 10; count > 0; count--)
    cout << count << end;
    cout << "Blast off!" << end;
```

Note: int declaration isn't required (as long as variable already was declared.)
Defining a function: example

Remember countdown function from 150?

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

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

**Ex:**

```c++
if (x < 0)
  x = -x;
```

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

- Don't need brackets if 1 line
- Don't need else
- No elseif
So meetings can get ugly!

```python
if ( )
    if ( )
        # code
    else
        # 3
if ( )
if ( )
    # 3
else
    # 3
```
Booleans & if/whiles

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

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

0 ⇔ false
Strange quirk in C++:

Consider

```cpp
int x;
x = 20;

if (x == 10)
    cout << "x is 10" << endl;
else
    cout << "x is not 10" << endl;
```