Data Structures

- Dr. Erin Wolf Chambers

Intro

Today

- Syllabus
- Dive in to C++

Resources you'll use:

- Webpage
- Recomended text
- cplusplus.com
- transition guide

This course:

Data structures in C++

First: data structures

What is a data structure?

Container to hold data

↳ along w/ specified ways to interact w/ data

Examples:

- Array ~ lists
- dictionaries
- matrices
- tree — search trees, heaps etc
- graph

Why should you care?

- You'll use them constantly!

There are many ways to solve a problem.

<u>Goals</u>:

① Correct

② Fast

③ Efficient

↳ Choice of data structure is key!

<u>Also</u>: job interviews!

Second: C++ (versus Python)

High vs low level

Python - English-like

more details in code

Interpreted vs. Compiled

command line running

2 phase process

Dynamic vs static typing

x = 5

⋮

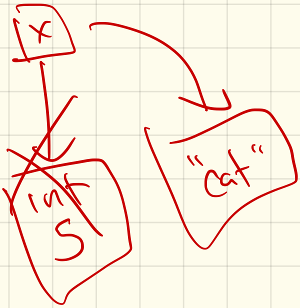
x = "cat"

int x = 5;

⋮

~~x = "cat";~~

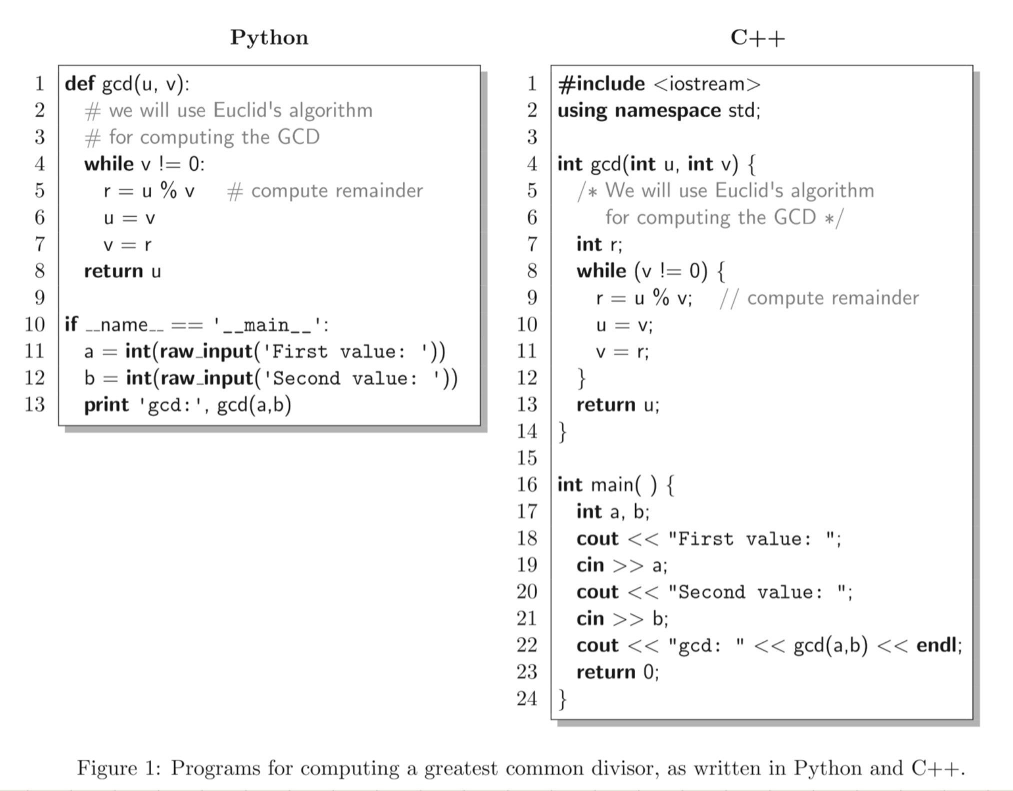
Error



Why should you learn C++?

- faster
- ubiquitous
- need to understand low level details (sometimes)
- more control

Comparison:



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++

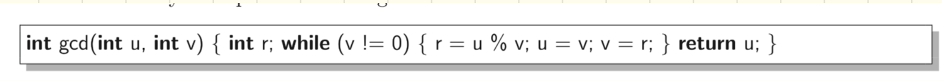
```cpp
#include <iostream>
using namespace std;

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++.

First: White space

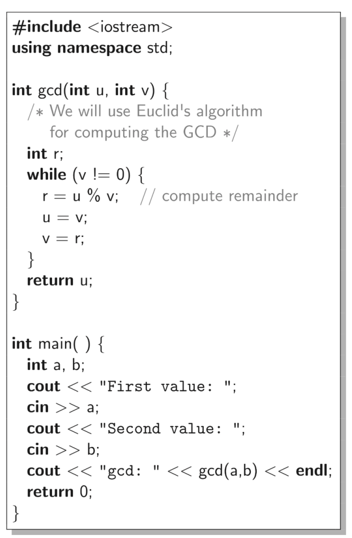
- returns, tabs, etc — **all** ignored in C++

(big difference from Python)



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

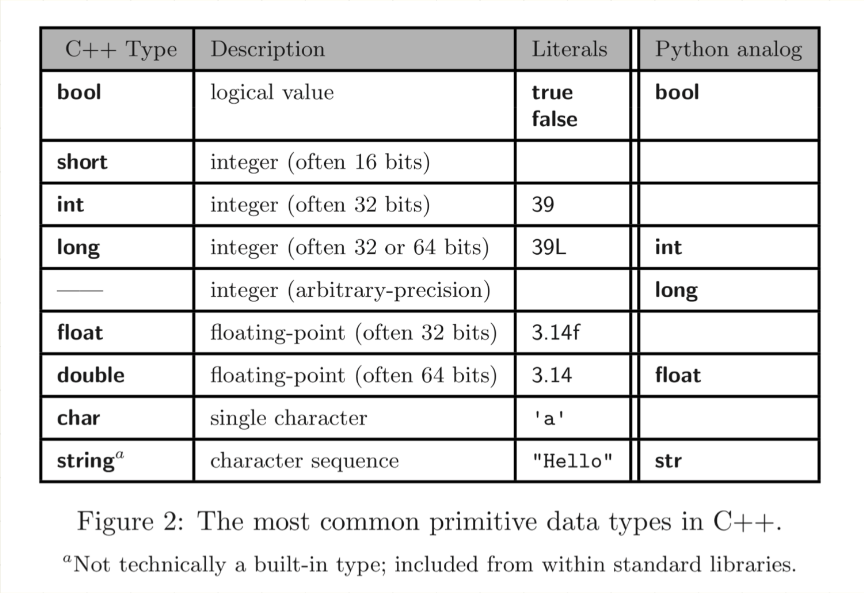
||



```
#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;
}
```

So control structures marked with ( ) and {},

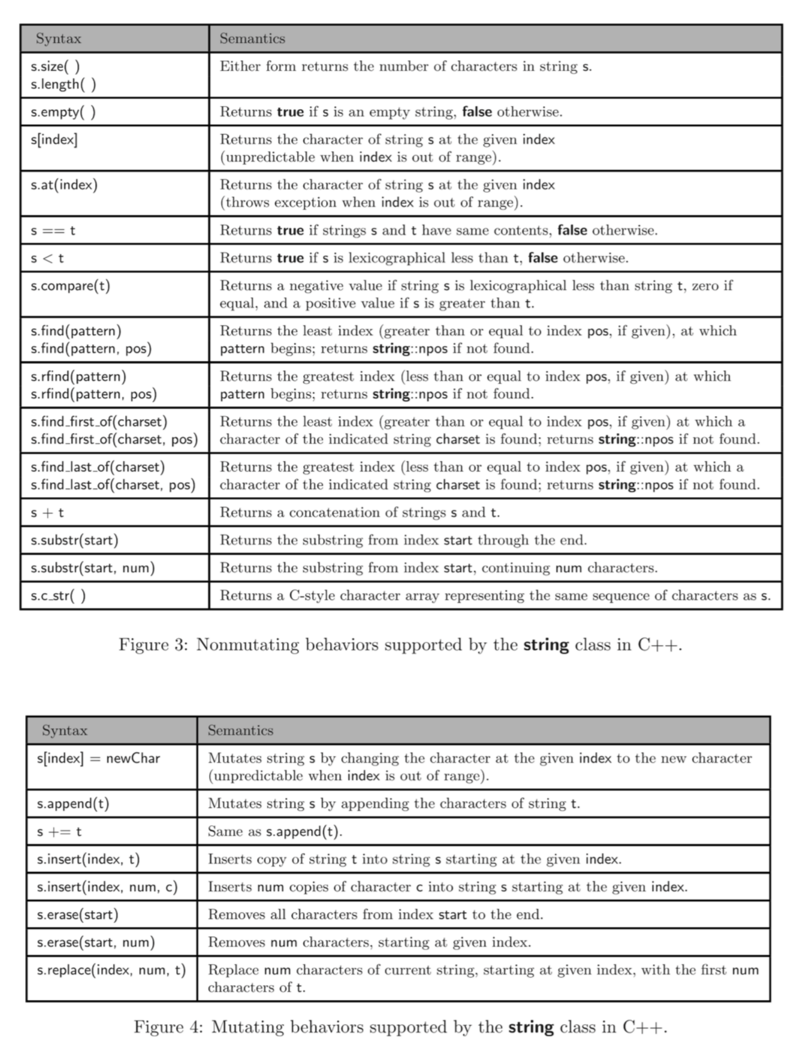
& lines end with ;



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.



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++.