



pollev.com/naomila



This stuff is hard.

**If you had to do it on a team (like for a job)
how would you like to be supported by your
team to become a better C++ programmer?**

CSE333 Systems Programming

C++: Standard Template Library

Instructors:

Naomi Alterman

Teaching Assistants:

Ann Baturytski

Barbora Buzkova

Mendel Carroll

Camden Harris

Hannah Hempstead

Rishabh Jain

Irene Lau

Jonathan Nister

Advay Patil

Aviel Wood


Angela Wu

Jennifer Xu

Administrivia

- ❖ Midterm on Monday in class!
- ❖ Review session tonight!
 - 5:30-7:30p in Gates G04, going over Winter 2026 midterm

C++'s Standard Library

- ❖ C++'s Standard Library consists of four major pieces:
 - 1) The entire C standard library
 - 2) C++'s input/output stream library
 - `std::cin`, `std::cout`, `stringstreams`, `fstreams`, etc.
 - 3) C++'s standard template library (STL) 
 - Containers, iterators, algorithms (sort, find, etc.), numerics
 - 4) C++'s miscellaneous library
 - Strings, exceptions, memory allocation, localization

STL Containers 😊

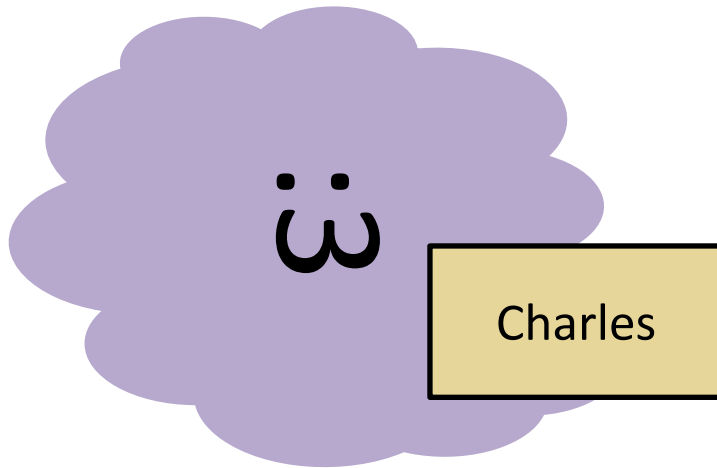
- ❖ A **container** is an object that stores (in memory) a collection of other objects (elements)
 - Implemented using class templates, so hugely flexible
 - More info in *C++ Primer* §9.2, 11.2
- ❖ Several different classes of container
 - Sequence containers (`vector`, `deque`, `list`, ...)
 - Associative containers (`set`, `map`, `multiset`, `multimap`, `bitset`, ...)
 - Differ in algorithmic cost and supported operations

STL Containers 🙄

- ❖ STL containers store by *value*, not by *reference*
 - When you insert an object, the container makes a *copy*
 - If the container needs to rearrange objects, it makes copies
 - *e.g.* if you sort a `vector`, it will make many, many copies
 - *e.g.* if you insert into a `map`, that may trigger several copies
 - What if you don't want this (disabled copy constructor or copying is expensive)?
 - You can insert a wrapper object with a pointer to the object
 - We'll learn about these “smart pointers” soon

Our Amoeba Class

- ❖ Toy class to examine how STL works
- ❖ A single instance represents a single Amoeba
- ❖ (with a name!!)

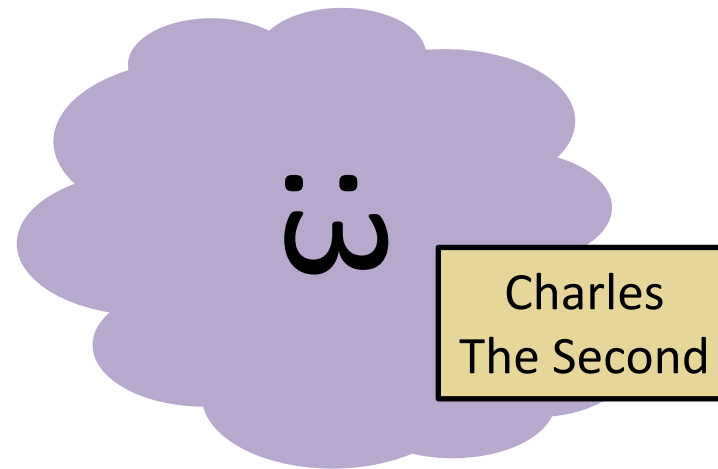
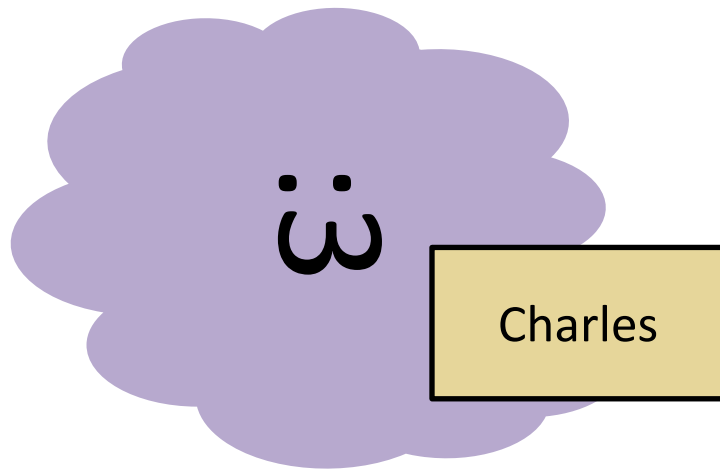


Charles lives (*and dies*) at a **fixed address in memory**

```
Amoeba c("Charles")
```

Our Amoeba Class

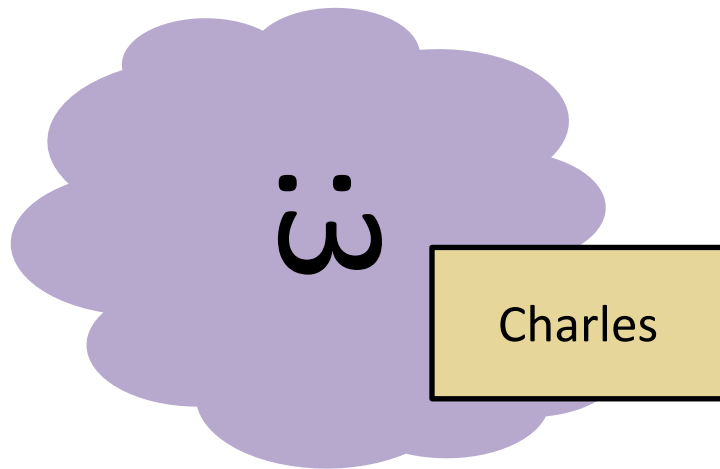
- ❖ The copy constructor means the Amoeba divided, and now has a child
- ❖ The child has the same name (but it's the next generation)



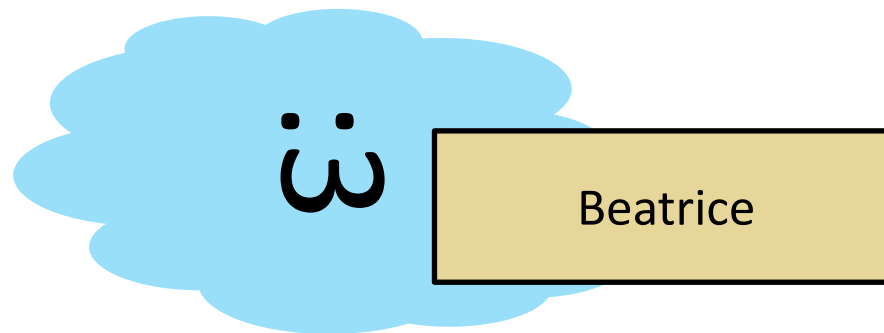
```
Amoeba c("Charles");  
Amoeba c2(c);
```

Our Amoeba Class

- ❖ The assignment operator means the left-hand Amoeba has been adopted by the right-hand Amoeba. It takes on the right-hand's name, and the right-hand's generation + 1. Just like a biological child would.
- ❖ But it also remembers its previous identity (roots are important!).

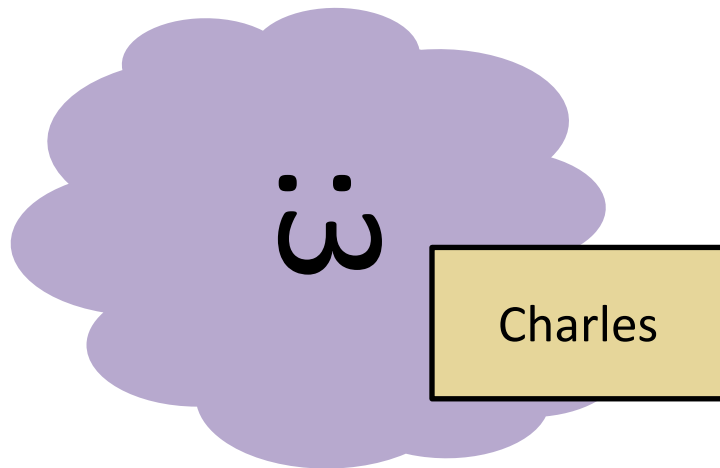


```
Amoeba c("Charles");  
Amoeba b("Beatrice");
```

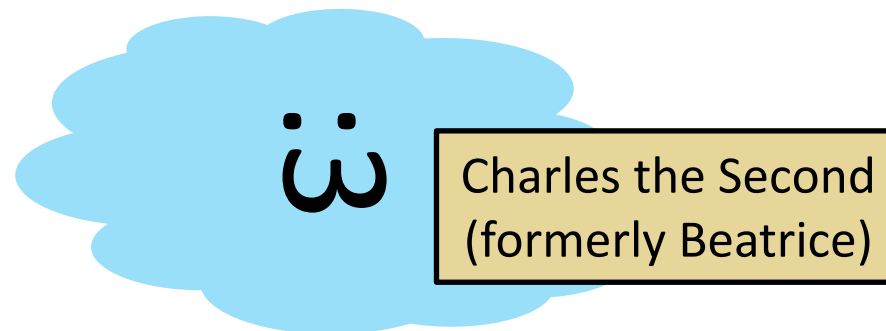


Our Amoeba Class

- ❖ The assignment operator means the left-hand Amoeba has been adopted by the right-hand Amoeba. It takes on the right-hand's name, and the right-hand's generation + 1. Just like a biological child would.
- ❖ But it also remembers its previous identity (roots are important!).



```
Amoeba c("Charles");  
Amoeba b("Beatrice");  
b = c;
```



STL `vector`

- ❖ A generic, dynamically resizable array
 - <http://www.cplusplus.com/reference/vector/vector/>
 - Elements are store in *contiguous* memory locations
 - Elements can be accessed using pointer arithmetic if you'd like
 - Random access is $O(1)$ time
 - Adding/removing from the end is cheap (amortized constant time)
 - Inserting/deleting from the middle or start is expensive (linear time)

vector/Amoeba Example

vectorfun.cc

```
#include <iostream>
#include <vector>
#include "Amoeba.h"

using namespace std;

int main(int argc, char** argv) {
    Amoeba a("Allen"), b("Barb"), c("Carol");
    vector<Amoeba> vec;

    cout << "vec.push_back " << a << endl;
    vec.push_back(a);
    cout << "vec.push_back " << b << endl;
    vec.push_back(b);
    cout << "vec.push_back " << c << endl;
    vec.push_back(c);

    cout << "vec[0]" << endl << vec[0] << endl;
    cout << "vec[2]" << endl << vec[2] << endl;

    return EXIT_SUCCESS;
}
```

Why All the Copying?

- ❖ What's going on here?
- ❖ Answer: a C++ vector (like Java's ArrayList) is initially small, but grows if needed as elements are added
 - Implemented by allocating a new, larger underlying array, copy existing elements to new array, and then replace previous array with new one
- ❖ And vector starts out *really* small by default, so it needs to grow almost immediately!
 - But you can specify an initial capacity if “really small” is an inefficient initial size (use “reserve” member function)
 - Example: see `vectorcap.cc`

STL `iterator`

- ❖ Each container class has an associated `iterator` class (e.g. `vector<int>::iterator`) used to iterate through elements of the container
 - <http://www.cplusplus.com/reference/iterator/iterator/>
 - `Iterator range` is from `begin` up to `end` i.e., `[begin, end)`
 - `end` is one past the last container element!
 - Some container iterators support more operations than others
 - All can be incremented (`++`), copied, copy-constructed
 - Some can be dereferenced on RHS (e.g. `x = *it;`)
 - Some can be dereferenced on LHS (e.g. `*it = x;`)
 - Some can be decremented (`--`)
 - Some support random access (`[]`, `+`, `-`, `+=`, `-=`, `<`, `>` operators)

iterator Example

vectoriterator.cc

```
#include <vector>

#include "Amoeba.h"

using namespace std;

int main(int argc, char** argv) {
    Amoeba a("Allen"), b("Barb"), c("Carol");
    vector<Amoeba> vec;

    vec.push_back(a);
    vec.push_back(b);
    vec.push_back(c);

    cout << "Iterating:" << endl;
    vector<Amoeba>::iterator it;
    for (it = vec.begin(); it < vec.end(); it++) {
        cout << *it << endl;
    }
    cout << "Done iterating!" << endl;
    return EXIT_SUCCESS;
}
```

Type Inference (C++11)

- ❖ The `auto` keyword can be used to infer types
 - Simplifies your life if, for example, functions return complicated types
 - The expression using `auto` must contain explicit initialization for it to work

```
for (vector<Amoeba>::iterator it = vec.begin(); it < vec.end(); it++) {  
    cout << *it << endl;  
}
```



```
for (auto it = vec.begin(); it < vec.end(); it++) {  
    cout << *it << endl;  
}
```

Range for Statement (C++11)

❖ Syntactic sugar similar to Java's `foreach`

■ General format:

```
for ( declaration : expression ) {  
    statements  
}
```

■ *declaration* defines loop variable

■ *expression* is an object representing a sequence

- Strings, initializer lists, arrays with an explicit length defined, STL containers that support iterators

```
// Prints out a string, one  
// character per line  
std::string str("hello");  
  
for ( auto c : str ) {  
    std::cout << c << std::endl;  
}
```

Updated iterator Example

vectoriterator_2011.cc

```
#include <vector>

#include "Amoeba.h"

using namespace std;

int main(int argc, char** argv) {
    Amoeba a("Allen"), b("Barb"), c("Carol");
    vector<Amoeba> vec;

    vec.push_back(a);
    vec.push_back(b);
    vec.push_back(c);

    cout << "Iterating:" << endl;
    for (auto & p : vec) { // p is a reference (alias) of vec elem
        cout << p << endl; // element here; not a new copy
    }
    cout << "Done iterating!" << endl;
    return EXIT_SUCCESS;
}
```

STL Algorithms

- ❖ A set of functions to be used on ranges of elements
 - **Range**: any sequence that can be accessed through *iterators* or *pointers*, like arrays or some of the containers
 - General form: `algorithm(begin, end, ...);`
- ❖ Algorithms operate directly on range *elements* rather than the containers they live in
 - Make use of elements' copy ctor, =, ==, !=, <
 - Some do not modify elements
 - e.g. find, count, for_each, min_element, binary_search
 - Some do modify elements
 - e.g. sort, transform, copy, swap

Algorithms Example

vectoralgos.cc

```
#include <vector>
#include <algorithm>
#include "Amoeba.h"
using namespace std;

void PrintOut(const Amoeba& p) {
    cout << " printout: " << p << endl;
}

int main(int argc, char** argv) {
    Amoeba a("Allen"), b("Barb"), c("Carol");
    vector<Amoeba> vec;

    vec.push_back(c);
    vec.push_back(a);
    vec.push_back(b);
    cout << "sort:" << endl;
    sort(vec.begin(), vec.end());
    cout << "done sort!" << endl;
    for_each(vec.begin(), vec.end(), &PrintOut);
    return EXIT_SUCCESS;
}
```

STL `list`

- ❖ A generic doubly-linked list
 - <http://www.cplusplus.com/reference/list/list/>
 - Elements are **not** stored in contiguous memory locations
 - Does not support random access (e.g. cannot do `list[5]`)
 - Some operations are much more efficient than vectors
 - Constant time insertion, deletion anywhere in list
 - Can iterate forward or backwards
 - Has a built-in sort member function
 - Doesn't copy! Manipulates list structure instead of element values

list Example

listexample.cc

```
#include <list>
#include <algorithm>
#include "Amoeba.h"
using namespace std;

void PrintOut(const Amoeba& p) {
    cout << " printout: " << p << endl;
}

int main(int argc, char** argv) {
    Amoeba a("Allen"), b("Barb"), c("Carol");
    list<Amoeba> lst;

    lst.push_back(c);
    lst.push_back(a);
    lst.push_back(b);
    cout << "sort:" << endl;
    lst.sort();
    cout << "done sort!" << endl;
    for_each(lst.begin(), lst.end(), &PrintOut);
    return EXIT_SUCCESS;
}
```

STL `map`

- ❖ One of C++'s *associative* containers: a key/value table, implemented as a search tree
 - <https://cplusplus.com/reference/map/map/>
 - General form: `map<key_type, value_type> name;`
 - Keys must be *unique*
 - `multimap` allows duplicate keys
 - Efficient lookup ($O(\log n)$) and insertion ($O(\log n)$)
 - Access value via `name[key]` or `.at(key)`
 - Elements are type `pair<key_type, value_type>` and are stored in *sorted* order (key is field `first`, value is field `second`)
 - Key type must support less-than operator (`<`)

map Example

mapexample.cc

```
void PrintOut(const pair<Amoeba, Amoeba>& p) {
    cout << "printout: [" << p.first << "," << p.second << "]" << endl;
}

int main(int argc, char** argv) {
    Amoeba a("Alvin"), b("Barb"), c("Chloe");
    Amoeba d("Daryl"), e("Ellen"), f("Fei");
    map<Amoeba, Amoeba> table;
    map<Amoeba, Amoeba>::iterator it;

    table.insert(pair<Amoeba, Amoeba>(a, b));
    table[c] = d;
    table[e] = f;
    cout << "table[e]:" << table[e] << endl;
    it = table.find(c);

    cout << "PrintOut(*it), where it = table.find(c)" << endl;
    PrintOut(*it);

    cout << "iterating:" << endl;
    for_each(table.begin(), table.end(), &PrintOut);

    return EXIT_SUCCESS;
}
```

Unordered Containers (C++11)

- ❖ `unordered_map`,
`unordered_set`
 - And related classes
`unordered_multimap`,
`unordered_multiset`
 - Average case for key access is $O(1)$
 - But range iterators can be less efficient than ordered `map/set`
 - See *C++ Primer*, online references for details



Extra Exercise #1

- ❖ Using the `Amoeba.h/.cc` files from lecture:
 - Construct a vector of lists of Amoebas
 - *i.e.* a `vector` container with each element being a `list` of Amoebas
 - Observe how many copies happen 😊
 - Use the sort algorithm to sort the vector
 - Use the `list.sort()` function to sort each list

Extra Exercise #2

- ❖ Take one of the books from HW2's `test_tree` and:
 - Read in the book, split it into words (you can use your `hw2`)
 - For each word, insert the word into an STL `map`
 - The key is the word, the value is an integer
 - The value should keep track of how many times you've seen the word, so each time you encounter the word, increment its map element
 - Thus, build a histogram of word count
 - Print out the histogram in order, sorted by word count
 - Bonus: Plot the histogram on a log-log scale (use Excel, gnuplot, etc.)
 - x-axis: $\log(\text{word number})$, y-axis: $\log(\text{word count})$