

C++ Standard Template Library

CSE 333 Winter 2020

Instructor: Justin Hsia

Teaching Assistants:

Andrew Hu

Austin Chan

Brennan Stein

Cheng Ni

Cosmo Wang

Diya Joy

Guramrit Singh

Mengqi Chen

Pat Kosakanchit

Rehaan Bhimani

Renshu Gu

Travis McGaha

Zachary Keyes

Administrivia

- ❖ Exercise 12 released today, due Monday
- ❖ Office hours today
 - 3:30 – 4:30, CSE2 153 (as normal)
 - 4:30 – 7:00, 4th floor breakout (extended)
- ❖ Midterm is next Friday (2/14) @ 5 - 6:10 pm in KNE 210/220
 - 1 double-sided page of handwritten notes; reference sheet provided on exam
 - **Topics:** everything from lecture, exercises, project, etc. up through **C++ classes and new/delete**
 - Old exams on course website, review in section next week

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 as class templates, so hugely flexible
 - More info in *C++ Primer* §9.2, 11.2
- ❖ Several different classes of container
 - Sequence containers (`vector`, `deque`, `list`, ...) *index numerically*
 - Associative containers (`set`, `map`, `multiset`, `multimap`, `bitset`, ...) *index by key*
 - 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 Tracer Class

- sets unique id_, initial value_ is id_-*
- ❖ Wrapper class for an `unsigned int` `value_`
 - Also holds unique `unsigned int` `id_` (increasing from 0)
 - Default ctor, cctor, dtor, `op=`, `op<` defined
 - `friend` function `operator<<` defined
 - Private helper method `PrintID()` to return
`"(id_, value_)"` as a string
 - Class and member definitions can be found in `Tracer.h` and
`Tracer.cc`
 - ❖ Useful for tracing behaviors of containers
 - All methods print identifying messages
 - Unique `id_` allows you to follow individual instances

STL `vector`

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

vector/Tracer Example

vectorfun.cc

```

#include <iostream>
#include <vector>      // most containers found in libraries of same name
#include "Tracer.h"

using namespace std;

int main(int argc, char** argv) {
    Tracer a, b, c;    // construct 3 Tracer instances
    vector<Tracer> vec; // new (empty) vector container of Tracers

    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;
}

```

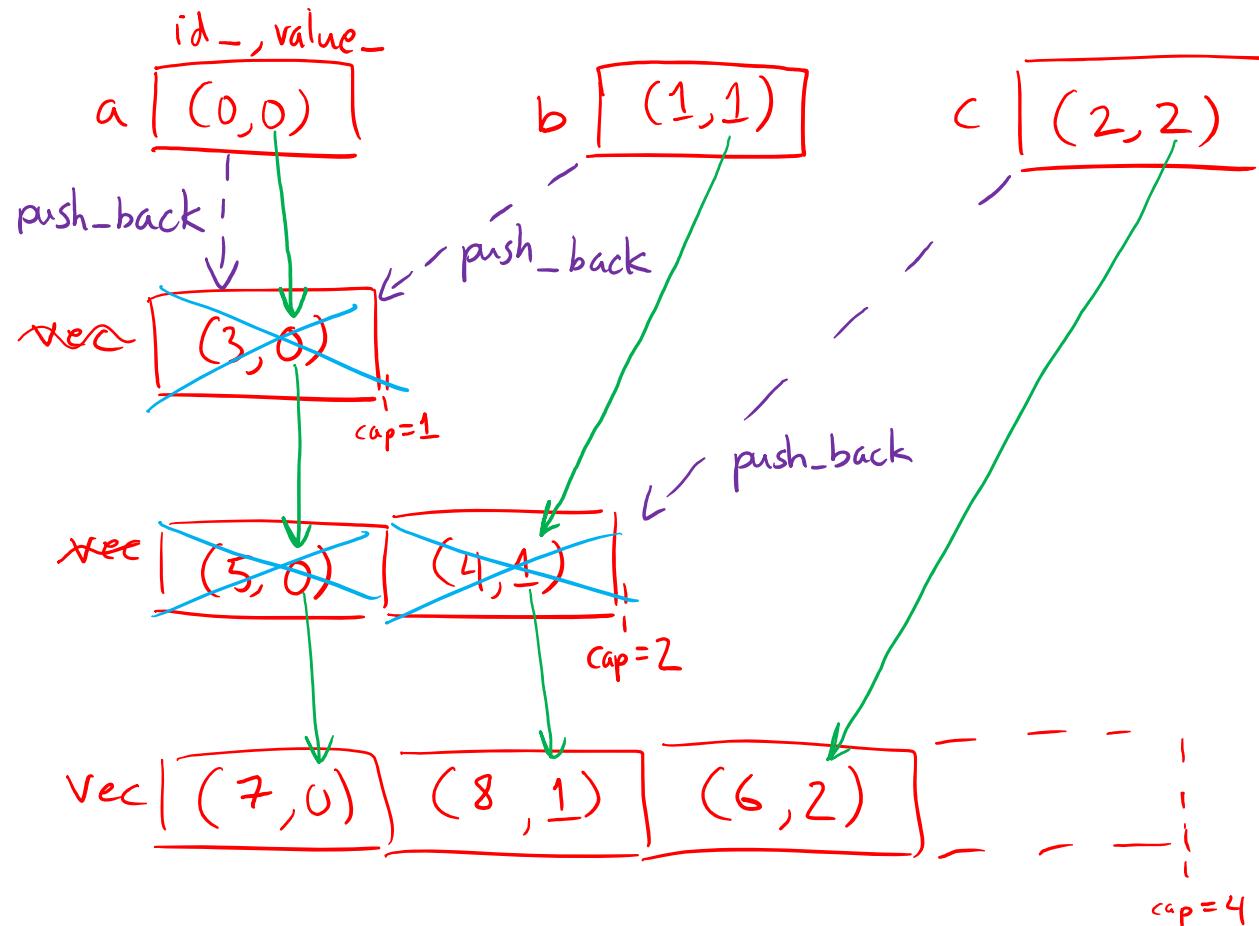
add copies of Tracers to end of container

elements can be accessed via subscript notation

verify the stored values are what we expect

Why All the Copying?

- copy construction
- destruction



push_back calls	Tracers constructed
0	3(a,b,c)
1	4
2	6
3	9
4	10
5	15

9 Tracer objects constructed!

Note: capacity doubles here each time (not an important detail)

Note: exact ordering of construction when vec gets moved not important

STL iterator

an iterator specific to the container & element type

- ❖ 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/std/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 "Tracer.h"

using namespace std;

int main(int argc, char** argv) {
    Tracer a, b, c;
    vector<Tracer> vec;

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

    cout << "Iterating:" << endl;
    vector<Tracer>::iterator it;
    for (it = vec.begin(); it < vec.end(); it++) {
        cout << *it << endl; // iterator of 1st element
    } // "dereference" to get element
    cout << "Done iterating!" << endl;
    return EXIT_SUCCESS;
}
```

iteratur one past last element

incrementing is always legal

iterator of 1st element

"dereference" to get element

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

```
// Calculate and return a vector
// containing all factors of n
std::vector<int> Factors(int n);

void foo(void) {
    // Manually identified type
    std::vector<int> facts1 =
        Factors(324234);

    // Inferred type
    auto facts2 = Factors(12321); compiler knows the return type of Factors()

    // Compiler error here
    auto facts3; ???
```

auto and Iterators

- ❖ Life becomes much simpler!

```
for (vector<Tracer>::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`

```
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 ( char c : str ) {  
    std::cout << c << std::endl;  
}
```

sequence of characters

Updated iterator Example

vectoriterator_2011.cc

```
#include <vector>

#include "Tracer.h"

using namespace std;

int main(int argc, char** argv) {
    Tracer a, b, c;
    vector<Tracer> vec;

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

    cout << "Iterating:" << endl;
    // "auto" is a C++11 feature not available on older compilers
    for (auto& p : vec) {
        cout << p << endl;
    }
    cout << "Done iterating!" << endl;
    return EXIT_SUCCESS;
}
```

greatly simplified!
iterator, begin, end handled for you

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*, ...);
Iterators defining a sequence
additional parameters depending on the algorithm
- ❖ Algorithms operate directly on range elements rather than the containers they live in
 - Make use of elements' copy ctor, =, ==, !=, < appropriate operator(s) must be defined for element type in order to use STL algorithms
 - 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 "Tracer.h"
using namespace std;

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

int main(int argc, char** argv) {
    Tracer a, b, c;
    vector<Tracer> vec;
    out of order
    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 0;
}
```

"initial" vec: | (? , 2) | (? , 0) | (? , 1) |

sort ↓

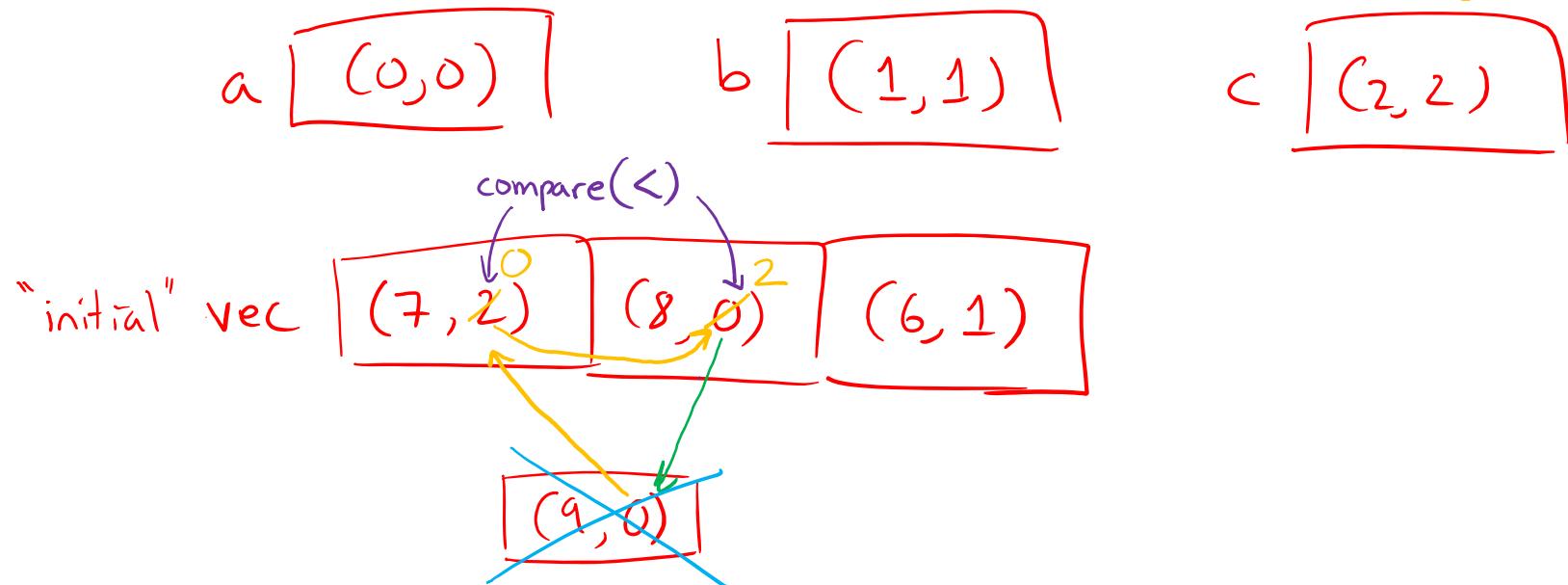
sorted vec: | (? , 0) | (? , 1) | (? , 2) |

Sorts elements in range

applies function to each element in range

Copying For sort

- copy construction
- destruction
- assignment operator



Note: only first comparison shown here.
more performed to complete swap() algorithm.

Iterator Question

- ❖ Write a function **OrderNext()** that takes a `vector<Tracer>` iterator and then does the compare-and-possibly-swap operation we saw in **sort()** on that element and the one *after* it
 - Hint: Iterators behave similarly to pointers!
 - Example: `OrderNext(vec.begin())` should order the first 2 elements of `vec`

```
void OrderNext(vector<Tracer>::iterator it1) {
```

```
    auto it2 = it1 + 1;  
    if (*it2 < *it1) {
```

```
        auto tmp = *it1;  
        *it1 = *it2;  
        *it2 = tmp;
```

vector<Tracer>::iterator
Tracer } }

Note: there are many equivalent implementations

Note: see the template version
(`vector<T>`) in `test.cc`

Extra Exercise #1

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