C++ Standard Template Library
CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:
Dao Yi            Farrell Fileas            Lukas Joswiak
Nathan Lipiarski  Renshu Gu               Travis McGaha
Yibo Cao          Yifan Bai               Yifan Xu
About how long did Homework 2 take?

A. 0-6 Hours
B. 6-12 Hours
C. 12-18 Hours
D. 18-24 Hours
E. 24+ Hours
F. I haven’t finished yet / I prefer not to say
Midterm in 1 week. Fri, Nov 1 – right here!
- You can bring ONE double-sided notes sheet; we’ll provide a reference sheet too
- Covers up to Wednesday (templates). So you can 😴 today!
- Old exams on course website
- Review in section next week

HW3 released on Monday, due Thu Nov 14
- Yes, that’s an entire 🎉 extra 7 days! 🎉
Lecture Outline

❖ Intro to Standard Template Library (STL)
❖ STL vector
❖ STL: Iterators
❖ STL: Algorithms
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* §9.1-2 (sequential), 11.1-2 (associative)

- 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 are defined in terms of their requirements, not by their implementation.
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?
  - *eg*, copying is expensive (Rule of 3!)
  - *eg*, you’ve disabled copies (Avoiding the Insanity!)
  - You can insert a wrapper object with a pointer to the “real” object
    - We’ll learn about these “smart pointers” soon
Our Tracer Class

- Contains a unique `int id_` (increasing from 0) and history of all its copies/assignments
- Definitions can be found in `Tracer.h` and `Tracer.cc`
  - Default ctor,cctor, dtor, `op=`, `op<` defined
  - `friend` function `operator<<` defined
  - Prints itself as "(id_, {previous ids})"
    - Last id in the list is its “original id”

- Useful for tracing behaviors of containers
  - All methods print identifying messages
  - Unique `id_` allows you to follow individual instances
Lecture Outline

- Intro to Standard Template Library (STL)
- **STL vector**
- STL: Iterators
- STL: Algorithms
**STL vector**

- **Requirement**: Random access is $O(1)$ time
  - Also: adding/removing from the end is cheap (amortized constant time)

- **Therefore**: A generic, dynamically resizable array
  - Corollaries:
    - Elements are stored in *contiguous* memory locations
    - Elements can be accessed using pointer arithmetic if you’d like
      - But it’s better style to use iterators
    - Inserting/deleting from the middle or start is expensive (linear time)
#include <iostream>
#include <vector>
#include "Tracer.h"

using namespace std;

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

    cout << "vec.push_back: " << a << endl;
    vec.push_back(a);  // inserts copies
    cout << "vec.push_back: " << b << endl;
    vec.push_back(b);
    cout << "vec.push_back: " << c << endl;
    vec.push_back(c);

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

    return EXIT_SUCCESS;
}
Why All the Copying?

i) vec: \emptyset

ii) vec: \begin{pmatrix} \text{vec: } (3, 3) \end{pmatrix}

iii) vec: \begin{pmatrix} \text{vec: } (5, 3, 0) \end{pmatrix}

iv) vec: \begin{pmatrix} \text{vec: } (7, 5, 3, 0) \end{pmatrix}

- Blue: copy
- vector impl. detail: default constructor for contained type is NOT required. This vector has capacity for one more instance, but has NOT called default constructor
- Total of 9 tracers created as a result of all the resizing
Lecture Outline

❖ Intro to Standard Template Library (STL)
❖ STL vector
❖ **STL: Iterators**
❖ STL: Algorithms
Each container class has an associated `iterator` class *(e.g. `vector<int>::iterator`)* used to iterate through elements of the container and also a `const_iterator`

- All can be incremented (``), copied, copy-constructed, and compared (`` , `!=``)

**Iterator range** is from `begin` up to `end`  
- i.e., `[begin, end)`. `end` is one past the last container element!
STL iterator (2 of 2)

- Some iterators support more operations than others
  - 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 functionality depends on the underlying container!
iterator Example

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

.end() is PAST the end, so we need < instead of <=

Iterators that don't define < can use "it != container.end()"
Type Inference (C++11)

❖ The **auto** keyword can be used to infer types
  ▪ Simplifies your life if, for example, using complicated types
  ▪ The expression using **auto** **must** be explicitly initialized so the compiler can make inference

```cpp
// 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 error here
  auto facts3;
}
```
auto and Iterators

- Life becomes much simpler!

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

```cpp
for (auto it = vec.begin(); it < vec.end(); it++) {
    cout << *it << endl;
}
```
Range-for Statement (C++11)

- Syntactic sugar similar to Java’s `foreach`

```cpp
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

```cpp
// Prints out a string, one character per line
std::string str("hello");

for ( auto c : str ) {
    std::cout << c << std::endl;
}
```
Updated iterator Example

```cpp
#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
    // Could also do for (auto p : vec)
    for (const auto& p : vec) {
        cout << p << endl;
    }
    cout << "Done iterating!" << endl;
    return EXIT_SUCCESS;
}
```

- **Example**: Shows how to use the `push_back` function with iterator and `for` loop.
- **Note**: The `auto` keyword is used for type deduction in C++11, which is not available on older compilers. The comment suggests an alternative method using a `for` loop with `auto` type deduction.
- **Hint**: The comment hints at the compiler wanting a `const_iterator` instead of a regular iterator for type safety and better performance.
Lecture Outline

❖ Intro to Standard Template Library (STL)
❖ STL vector
❖ STL: Iterators
❖ **STL: Algorithms**
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 most 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

```cpp
#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;
    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;
}
```
Copying For sort

Unsorted vec:

(7, 323) (8, 303) (6, 313)

(9, 383)

(a | (0, 33))

(b | (1, 33))

(c | (2, 33))

Note: only the first comparison shown here, more comparisons + swaps necessary to sort entire vec
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`
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 Tracers
  - Observe how many copies happen 😊
    - Use the sort algorithm to sort the vector
    - Use the `list.sort()` method to sort each list