C++ Standard Template Library
CSE 333 Spring 2018

Instructor: Justin Hsia

Teaching Assistants:
Danny Allen       Dennis Shao       Eddie Huang
Kevin Bi          Jack Xu           Matthew Neldam
Michael Poulain   Renshu Gu        Robby Marver
Waylon Huang      Wei Lin
 Administrivia

- No exercise released today!

- Homework 2 due tomorrow (4/26)
  - Don’t forget to clone your repo to double-/triple-/quadruple-check compilation!

- Midterm is next Friday (5/4) @ 5–6 pm in GUG 220
  - 1 double-sided page of hand-written notes; reference info may be provided on exam (subject to change)
  - Topics: everything from lecture, exercises, project, etc. up through hw2 and C++ 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
Container Drawbacks

- 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

- Wrapper class for an unsigned int value_
  - Default ctor, cctor, dctor, op=, op< defined
  - friend function operator<< defined
  - Also holds unique unsigned int id_ (increasing from 0)
  - 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 will print identifying messages
  - Unique id_ allows you to follow individual instances
STL vector

- A generic, dynamically resizeable array
  - Elements are stored in contiguous memory locations
    - Elements can be accessed using pointer arithmetic if you’d like to
    - Random access is $O(1)$ time
  - Adding/removing from the end is cheap (constant time)
  - Inserting/deleting from the middle or start is expensive (linear time)
    - must copy all following elements
vector/Tracer Example

```cpp
#include <iostream>
#include <vector>
#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;
    return 0;
}
```

// most containers found in libraries of same name

// 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?

Note: capacity doubles here each time (not an important detail)
Note: exact ordering of construction when Vec gets moved not important
STL iterator

Each container class has an associated iterator class (e.g. `vector<int>::iterator`) used to iterate through elements of the container

- Iterator range is from `begin` up to `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)
#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 (auto it = vec.begin(); it < vec.end(); it++) {
        cout << *it << endl;
        // "dereference" to get element
    }
    cout << "Done iterating!" << endl;
    return 0;
}
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

```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;
}

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

- Syntactic sugar that emulates 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
    for (auto& p : vec) { // greatly simplified!
        cout << p << endl;
    }
    cout << "Done iterating!" << endl;
    return 0;
}
```
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

```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;
}
```

vectoralgos.cc
Copying For sort

```
{0,0}  {1,1}  {2,2}
```

```
{7,2}  {8,0}  {6,1}
```

```
{9,0}  
```

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`.

```cpp
void OrderNext(vector<Tracer>::iterator it1) {
    auto it2 = it1 + 1;
    if (*it2 < *it1) {
        auto tmp = *it1;
        *it1 = *it2;
        *it2 = tmp;
    }
}
```

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 Tracers*
  - Observe how many copies happen 🤪
    - Use the sort algorithm to sort the vector
    - Use the `list.sort()` function to sort each list