C++ Standard Template Library
CSE 333

Instructor: Hannah C. Tang

Teaching Assistants:
Deeksha Vatwani  Hannah Jiang  Jen Xu
Leanna Nguyen  Nam Nguyen  Sayuj Shahi
Tanay Vakharia  Wei Wu  Yiqing Wang
Zohar Le
Recall our templated `compare()` function from last lecture. What happens if we instantiate it on a type that doesn't support `operator<`?

```cpp
#ifndef COMPARE_H_
define COMPARE_H_

template <typename T>
int comp(const T& a, const T& b) {
 if (a < b) return -1;
 if (b < a) return 1;
 return 0;
}

#include "compare.h"

int main(int argc, char **argv) {
 int i = comp<vector<int>>(1, 2);
 return EXIT_SUCCESS;
}
```
Administrivia

❖ ex11 released Friday, but due on Wednesday so that we can answer questions!

❖ HW3 spec posted later today
  ▪ Starter code will be pushed to repos tomorrow
  ▪ Short demo today or early next week depending on time available
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*, ...)
  - **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 Tracer Class

❖ Wrapper class for an int value_
  ▪ Default ctor, cctor, dtor, op=, op< defined
  ▪ friend function operator<< defined
  ▪ Also holds unique 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 print identifying messages
  ▪ Unique id_ allows you to follow individual instances
**STL vector**

- A generic, dynamically resizable array
  - Elements are stored 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/Tracer Example

```cpp
#include <iostream>
#include <vector>
#include "Tracer.h"

using namespace std;

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

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

    cout << "\nvec[0]" << endl << vec[0] << 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
  ▪ **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)
#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;
    }
    cout << "Done iterating!" << endl;
    return EXIT_SUCCESS;
}
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;
}
```

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

- Syntactic sugar similar to Java’s `for each`

```c++
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

```c++
// 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
t #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 & p : vec) { // p is a reference (alias) of vec
        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

```cpp
#include <vector>
#include <algorithm>
#include "Tracer.h"

using namespace std;

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

template<typename T>

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 EXIT_SUCCESS;
}
```

vectoralgos.cc
STL list

- A generic doubly-linked 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

```cpp
#include <list>
#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;
    list<Tracer> 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
  - 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]`
  - 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

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

int main(int argc, char** argv) {
    Tracer a, b, c, d, e, f;
    map<Tracer, Tracer> table;
    map<Tracer, Tracer>::iterator it;

    table.insert(pair<Tracer, Tracer>(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 `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
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(word number), y-axis: log(word count)