C++ Standard Template Library
CSE 333 Winter 2025

Instructor: Hal Perkins

Teaching Assistants:

Lainey Jeon	Hannah Jiang	Irene Lau
Nathan Li	Janani Raghavan	Sean Siddens
Deeksha Vatwani	Yiqing Wang	Wei Wu
Jennifer Xu		

Administrivia (added Friday)

- ❖ New exercise ex11 out today. Due Monday morning.
  - ▪ uses STL vector and sort

- ❖ HW3 description posted now
  - ▪ Starter code will be pushed to repos tomorrow
  - ▪ Short demo today or early next week depending on time available

- ❖ Midterm exam next Thursday, 2/13, 5-6 pm, Kane 110
  - ▪ Topic list and old exams & topic list on website
  - ▪ Closed book, slides, etc., but you may have one 5x8 notecard with whatever handwritten notes you want on both sides
    - • Free blank cards available in class this week and next ☺
  - ▪ Review in sections next week – bring questions!!

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

  ▪ <u>Sequence</u> containers (`vector, deque, list, ...`)

  ▪ <u>Associative</u> 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

- ▪ http://www.cplusplus.com/reference/stl/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/Tracer Example

vectorfun.cc

```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 << "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/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

```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;
  vector<Tracer>::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

```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!

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

14

Updated `iterator` Example

vectoriterator_2011.cc

```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 (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

vectoralgos.cc

```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 EXIT_SUCCESS;
}
```

STL `list`

- ❖ A generic doubly-linked list
    - http://www.cplusplus.com/reference/stl/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

```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;
}
```

20

STL `map`

❖ One of C++'s *associative* containers: a key/value table, implemented as a search tree

- http://www.cplusplus.com/reference/stl/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]`

- 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

```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

If we get here on Friday…

❖ Done with templates, STL, etc.

❖ Vector exercise (ex11) due Monday

❖ Next exercise (ex12) uses maps

  ▪ Due next Wednesday

  ▪ Release now so everyone can work on it over the weekend if they want? (Might be helpful since midterm exam is next Thur.)

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

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

- <u>Bonus</u>: Plot the histogram on a log-log scale (use Excel, gnuplot, etc.)

  - x-axis: log(word number), y-axis: log(word count)