C++ STL (part 2 of 2)

CSE 333 Spring 2023

Instructor: Chris Thachuk

Teaching Assistants:

Byron Jin

Deeksha Vatwani

Humza Lala

Noa Ferman

Seulchan (Paul) Han

Tim Mandzyuk

CJ Reith

Edward Zhang

Lahari Nidadavolu

Saket Gollapudi

Timmy Yang

Wui Wu

Relevant Course Information

- Homework 3 will be released today, due in ~***3 weeks***

- Midterm: May 4 – May 6 (1pm)
  - Take home (Gradescope) and open notes
  - Individual, but high-level discussion allowed ("Gilligan's Island Rule")
  - No lecture Friday (May 5); I'll be in lecture room to answer questions

vector/Tracer Example

vectorfun.cc

```
#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?

STL `iterator`

- Each container class has an associated iterator class (*e.g.*, `vector<int>::iterator`) used to iterate through elements of the container
  - https://cplusplus.com/reference/iterator/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;
  }
  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

```
// 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;
}
```

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;
}
```

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

```
#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`

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`

Lecture Outline

- STL iterators, algorithms
- **STL (finish)**
  - **List**
  - **Map**

STL `list`

- A generic doubly-linked list
  - https://cplusplus.com/reference/list/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

```
#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
  - https://cplusplus.com/reference/map/map/
  - General form: `map<key_type, value_type> name;`
  - Keys must be *unique*
    - `multimap` allows duplicate keys
  - Efficient lookup ($\mathcal{O}(\log n)$) and insertion ($\mathcal{O}(\log n)$)
    - 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

```
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;
}
```

Basic map Usage

- animals.cc

Basic `map` Usage

- `animals.cc`

  - https://www.youtube.com/watch?v=jofNR_WkoCE

Homegrown `pair<>`

Unordered Containers (C++11)

- `unordered_map`, `unordered_set`
  - And related classes `unordered_multimap`, `unordered_multiset`
  - Average case for key access is $\mathcal{O}(1)$
    - But range iterators can be less efficient than ordered `map`/`set`
  - See *C++ Primer*, online references for details