C++ STL & Mid Quarter Review
CSE 333 Summer 2020

Instructor: Travis McGaha

Teaching Assistants:
Jeter Arellano  Ramya Challa  Kyrie Dowling
Ian Hsiao     Allen Jung    Sylvia Wang
About how long did Homework 2 take?

A. 0-4 Hours  
B. 4-8 Hours  
C. 8-12 Hours  
D. 12-16 Hours  
E. 16-20 Hours  
F. 20+ Hours  
G. I didn’t submit / I prefer not to say

Side question: how are you liking C++?
Administrivia

- Exercise 12 released today, due Monday

- Mid Quarter Survey due Monday (7/27) @ 11:59 pm
  - Feedback will be used to try and better the rest of this quarter and future quarters!

- Homework 3
  - to be pushed out later tonight or tomorrow.
  - Due Thursday (8/6) @ 11:59 pm
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
    - 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)

Like a normal C array!

Need to shift all of the elements in the array
**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 (`[ ]`, `+`, `-`, `+=`, `-=`)
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;
    vector<Tracer>::iterator it;
    for (it = vec.begin(); it < vec.end(); it++) {
        cout << *it << endl;
    }
    cout << "Done iterating!" << endl;
    return EXIT_SUCCESS;
}
```

vectoriterator.cc
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;
}
```

Look at all this space!!!

Another beautiful feature of C++ 😊
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
    for (auto& p : vec) {
        cout << p << endl;
    }
    cout << "Done iterating!" << endl;
    return EXIT_SUCCESS;
}
```

Look at how much more simplified this is! No `begin()`, `end()`, or dereferencing! :O
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, ...);`
    Takes a range of a sequence to operate on
  
  ❖ 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;
}
```

Not in order 😊

Sort elements from [vec.begin(), vec.end()]

Runs function on each element.
In this case, prints out each element
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

Iterate backward: `--`
Iterate forward: `++`
#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 0;
}
```

Use case is similar to Vector, but internal implementation is different

Won’t copy elements, just modifies the next and prev pointers
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]`
      - if key doesn’t exist in map, it is added to the map
  - 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
#include <map>

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);  // Returns iterator. (end if not found)
    cout << "PrintOut(*it), where it = table.find(c)" << endl;
    PrintOut(*it);

    cout << "iterating:" << endl;
    for_each(table.begin(), table.end(), &PrintOut);

    return 0;
}
```

### Map elements

- Equivalent behavior
  - Returns iterator. (end if not found)
  - can also use map.count() to see if a key exists
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
We don’t have a midterm this quarter

For the rest of lecture, we will do some conceptual questions to reflect on what we learned in the first half of the course.

- Hopefully exercises & HW gave you good C/C++ coding experiences 😊
Should we use a reference?

A. We must NOT use a reference
B. It’s OK but discouraged to use a reference
C. It’s OK and encouraged to use a reference
D. We must use a reference
E. We’re lost…

```cpp
#ifndef _COMPLEX_H_
#define _COMPLEX_H_

#include <iostream>

namespace complex {

class Complex {
    public:
        // Copy constructor, should we pass a reference or not? (Answer: ?)
        Complex(const Complex &copyme) {
            real_ = copyme.real_;  // Copy real part
            imag_ = copyme.imag_;  // Copy imaginary part
        }

    private:
        double real_, imag_;  // Use double for complex numbers
    }

} // namespace complex

#endif // _COMPLEX_H_
```
Complex1.h

D. We must use a reference
   - A const reference to a complex type
   - We aren’t changing the argument’s values so it doesn’t matter if we use a copy or not, in theory
   - A copy constructor must take a reference, otherwise it would need to call itself to make a (call-by-value) copy of the argument...
Should we use a reference?

A. We must NOT use a reference
B. It’s OK but discouraged to use a reference
C. It’s OK and encouraged to use a reference
D. We must use a reference
E. We’re lost...

```cpp
#include <iostream>

namespace complex {

class Complex {
  public:
    // Should operator+ return a reference or not?
    // (Answer: ?)
    Complex &operator+(const Complex &a) const {
      Complex tmp(0, 0);
      tmp.real_ = this->real_ + a.real_;
      tmp.imag_ = this->imag_ + a.imag_;
      return tmp;
    }

  private:
    double real_, imag_;  // This line is wrong; fix it.
};  // class Complex
}  // namespace complex
```
Complex2.h

A. We must NOT use a reference

- A reference to a stack-allocated complex type
- Never return a reference (or pointer to) a local variable
  - Destructor is also called on object when returning
Provided are three different ways to read the contents of a file. Rank the implementations by their efficiency.

- Assume that buffers are allocated and files opened/closed for you

**Implementation #1**

```cpp
fread(buf, LEN, sizeof(char), file);
return buf;
```

**Implementation #2**

```cpp
while(read(fd, buf+numread, sizeof(char)) != 0) {
    // ...
    numread += sizeof(char);
}
return buf;
```

**Implementation #3**

```cpp
while((res = read(fd, buf+numread, LEN - numread) != 0) {
    // ...
    numread += res;
}
return buf;
```
Provided are three different ways to read the contents of a file. Rank the implementations by their efficiency.

3 > 1 >>>>> 2

Implementation #1

```c
fread(buf, LEN, sizeof(char), file);
return buf;
```

Implementation #2

```c
while(read(fd, buf+numread, sizeof(char)) != 0) {
    // ...  
    numread += sizeof(char);
}
return buf;
```

Implementation #3

```c
while((res = read(fd, buf+numread, LEN - numread) != 0) {
    // ...  
    numread += res;
}
return buf;
```

Buffered read, minimal system calls, but copies contents twice. Once to internal buffer, then to buf.

One system call per character! SYSTEM CALLS TAKE A LONG TIME

Reads in as much as possible per system call, no double copying via buffer.
What is wrong with this program?

(ignoring style issues)

```c
#define FOO 333
struct pair {
    int x, y;
}
#include "pair.h"
#include "util.h"

#include "stdio.h"

void Pair_Allocate(pair *out) {
    out = (pair *) malloc(sizeof(pair))
    out->x = 0;
    out->y = 0;
}

#include "pair.h"
#include "util.h"

int main() {
    pair * p;
    Pair_Allocate(p);
    p->x = FOO;
    p->y = 351;
    Pair_Print(*p);
}

#include "pair.h"
#include <stdio.h>

void Pair_Allocate(pair *out) {
    out = (pair *) malloc(sizeof(pair))
    out->x = 0;
    out->y = 0;
}

void Pair_Print(pair *p) {
    printf("(x:%d, y:%d)", p.x, p.y);
}
```
What is wrong with this program?

- (ignoring style issues)

```c
#define FOO 333
struct pair {
    int x, y;
};

#include "pair.h"
#include <stdio.h>

void Pair_Allocate(pair *out) {
    out = (pair *) malloc(sizeof(pair));
    out->x = 0;
    out->y = 0;
}

#include "pair.h"
#include "util.h"

void Pair_Print(pair *p) {
    printf("(x:%d, y:%d)", p.x, p.y);
}

int main() {
    pair * p;
    Pair_Allocate(p);
    p->x = FOO;
    p->y = 351;
    Pair_Print(*p);
}
```
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)