

THE C++ STL

LECTURE 11-1

JIM FIX, REED COLLEGE CS2-S20

2ND MIDTERM THURSDAY

- ▶ **Midterm** Thursday-Friday on circuits and MIPS assembly
 - Will post PDF in Git repo on as an assignment link **Thursday by noon PST**
 - Designed as an in-class **80 minute exam** with 6 problems
 - Submit images and/or text files within repo on GitHub
 - Use **about 2 hours** total to take the exam and to assemble/submit work
 - Must be submitted **Friday by noon PST**
 - Treat as a written exam. E.g. don't need to compile/test.

THE C++ STANDARD TEMPLATE LIBRARY (STL)

- ▶ A large collection of fully-realized C++ (templated) classes.
- ▶ Provides a lot of useful data types and functions.
 - container classes:
 - ♦ **vector, array**
 - ♦ **stack, queue, priority_queue**
 - ♦ **map, unordered_map**
 - **Iterator** for traversing these data structures
 - ➔ iterators are a generalized traversing pointer (or "handle")
 - **#include <algorithm>** for sorting, permuting, ...
 - ♦ supported by *lambdas*
 - "smart" pointers that provide better sharing and memory management

WHY USE THESE?

- ▶ Highly optimized
- ▶ Mostly generic– carefully designed for lots of uses, contexts
- ▶ Safe, debugged;
- ▶ Supported by future evolution of the language
- ▶ Adopted by modern C++ programmers
 - Using them, your code will make sense to others

A WORD ON MY PEDAGOGICAL CONFLICT

► This course had competing goals:

- **OTOH.** Wanted to teach you low-level details...
 - ✦ machine representation: bits, bytes, assembly, GOTOs, ...
 - ✦ linking/arrays and memory organization
 - ✦ what's underneath standard data structures ("roll our own")
- **OTOH.** Aspire to teach you how to engineer maintainable, readable, code
 - ✦ Mastering the STL might be the better approach to learning C++
 - Can write more Pythonic code; but with compiled performance

► **So maybe...**

Mastering C++ requires you to learn the "raw" stuff, but shouldn't use it.

AND SO...

- ▶ C++, for me, was a way of introducing you to low-level stuff
- ▶ But others actually use it to engineer **maintainable, readable, correct** code
 - Requires years of practice within C++ (and other languages, as well)
 - Using the C++ STL well is part of that practice
- ▶ **NOTE:** probably shouldn't be using C-style "raw" arrays
 - The **vector** and **array** types in STL were intended to replace them
- ▶ **NOTE:** probably shouldn't roll your own data structures
 - there are a wealth of STL ones for most common ones
- ▶ **NOTE:** probably should only use pointers/linking/etc. sparingly
 - learn smart pointer classes **shared_ptr** and **weak_ptr**

TODAY: LOOK AT VECTOR

- Here is a simple example of its use

```
#include <vector>
...

std::vector<int> iv {7,1,3,4,8};

// Output the elements using a "for" over the vector.
for (int x : iv) {
    std::cout << x << "\n";
}
std::cout << std::endl;

// Sum the elements using a "for" over the vector.
int sum = 0;
for (int x : iv) {
    sum += x;
}
std::cout << sum << std::endl;
```

FOR LOOP WITH AN ELEMENT REFERENCE

- Can also get a reference to each vector element

```
std::vector<int> iv {7,1,3,4,8};

// Update the elements using a "for" over the vector.
for (int& e : iv) { // Note the use of & here
    e = e + 10;
}

// This actually adds 10 to each vector component.
```


OPERATORS THAT LOOK LIKE ARRAY ACCESS

- Can use **operator[]** just like you can with a C-style array:

```
std::vector<int> iv {7,1,3,4,8};

// Output the elements by accessing each by an index.
for (int i = 0; i < iv.size(); i++) {
    std::cout << iv[i] << "\n";
}
std::cout << std::endl;

// Add 10 to each element
for (int i = 0; i < iv.size(); i++) {
    iv[i] = iv[i] + 10;
}
```

ITERATOR SYNTAX

► Can instead use iterators

```
std::vector<int> iv {7,1,3,4,8};

for (std::vector<int>::iterator p = iv.begin();
     p != iv.end();
     ++p) {
    std::cout << (*p) << "\n";
}
std::cout << "\n";

for (std::vector<int>::iterator p = iv.begin();
     p != iv.end();
     ++p) {
    (*p) += 10;
}
```

ITERATOR SYNTAX

► Can instead use iterators

```
std::vector<int> iv {7,1,3,4,8};

for (std::vector<int>::iterator p = iv.begin();
     p != iv.end();
     ++p) {
    std::cout << (*p) << "\n";
}
std::cout << "\n";

for (std::vector<int>::iterator p = iv.begin();
     p != iv.end();
     ++p) {
    (*p) += 10;
}
```

ITERATOR SYNTAX (DECLARED INSTEAD WITH **AUTO**)

- ▶ Have I told you yet about **auto**????

```
std::vector<int> iv {7,1,3,4,8};
```

```
for (auto p = iv.begin(); p != iv.end(); ++p) {  
    std::cout << (*p) << "\n";  
}  
std::cout << "\n";
```

```
for (auto p = iv.begin(); p != iv.end(); ++p) {  
    (*p) += 10;  
}
```

- ▶ The **auto** keyword lets C++ *infer* the type of the variable.
 - Some consider it good style (btw I **LOVE** *type inference* in *other* languages)
 - *Please* continue to write explicit types in CS2 C++ until the semester ends

ITERATORS TIE NICELY WITH <ALGORITHM>

- ▶ Here is a sorting library function

```
#include <algorithm>
```

```
...
```

```
sort(iv.begin(), iv.end(), std::greater<int>());
```

- ▶ We're passing iterators (pointer-ish thing) within a vector:
 - one for the beginning, one for the ending element
 - gives the *extent* in a contiguous container
- ▶ Third argument is a function for comparing two **int** values
 - from Stroustrup example; sorts in reverse

THE **AT** METHOD PERFORMS "BOUNDS CHECKING"

- ▶ **NOTE: `operator[]`** does not check the index.
- ▶ To have the class perform bounds checking use **`at`** instead.

```
std::vector<int> iv {7,1,3,4,8};

// Output the elements by accessing each by an index.
for (int i = 0; i < iv.size(); i++) {
    std::cout << iv.at(i) << "\n";
}
std::cout << std::endl;

// Add 10 to each element
for (int i = 0; i < iv.size(); i++) {
    iv.at(i) = iv.at(i) + 10;
}
```

GROWING A VECTOR

- ▶ C++ programmers often **push_back**

```
std::vector<int> iv {};  
std::string entry;  
do {  
    std::cin >> entry;  
    if (entry != "done") {  
        int value = std::stoi(entry);  
        iv.push_back(value); // puts at the end of the vector  
    } while (entry != "done");
```

- ▶ Under the covers, C++ is resizing occasionally, maintaining a C-style array.

GROWING A VECTOR; SHRINKING A VECTOR

- ▶ C++ programmers often **push_back**

```
std::vector<int> iv {};  
std::string entry;  
do {  
    std::cin >> entry;  
    if (entry != "done") {  
        int value = std::stoi(entry);  
        iv.push_back(value); // puts at the end of the vector  
    } while (entry != "done");
```

- ▶ Under the covers, C++ is resizing occasionally, maintaining a C-style array.
- ▶ There is also a method **pop_back**
 - ➡ This shrinks the **vector**, and the last element is removed.

RESIZING A VECTOR

- ▶ You can do several other things, for example **resize**

```
iv.resize(12);
```

- ▶ If size is 5, then this performs a chunk of 7 "push backs"
 - it fills those extra 7 elements with the default/"zero" value
- ▶ **NOTE:** different than **reserve**, which adds capacity *under the covers*

```
iv.reserve(new_capacity);
```

EDITING WITHIN A VECTOR

- ▶ You can perform "iterator arithmetic" to work within a vector:

```
std::vector<int>::iterator place = iv.begin()+6;  
(*place) = 4567890;
```

- ➡ This modifies the item at index 6

EDITING WITHIN A VECTOR

- ▶ You can perform "iterator arithmetic" to work within a vector:

```
std::vector<int>::iterator place = iv.begin()+6;  
(*place) = 4567890;
```

- ➡ This modifies the item at index 6

EDITING WITHIN A VECTOR

- ▶ You can perform "iterator arithmetic" to work within a vector:

```
std::vector<int>::iterator place = iv.begin()+6;  
(*place) = 4567890;
```

→ This modifies the item at index 6

- ▶ Can **insert** a new item, say, before the one at index 4

```
iv.insert(iv.begin()+4, 987);
```

- ▶ Can **erase** a chunk of items (like Python's **[2:-4]** list range notation)

```
iv.erase(iv.begin()+2, iv.end()-4);
```

EDITING WITHIN A VECTOR

- ▶ You can perform "iterator arithmetic" to work within a vector:

```
std::vector<int>::iterator place = iv.begin()+6;  
(*place) = 4567890;
```

→ This modifies the item at index 6

- ▶ Can **insert** a new item, say, before the one at index 4

```
iv.insert(iv.begin()+4, 987);
```

- ▶ Can **erase** a chunk of items (like Python's `[2:-4]` list range notation)

```
iv.erase(iv.begin()+2, iv.end()-4);
```

VECTOR STORAGE

- ▶ I did a little test to see how storage is managed. Consider this class

```
class Box {  
public:  
    int value;  
    Box(int v) : value {v} { }  
    void square() { value *= value; }  
}
```

- ▶ Consider this client code

```
Box a {4};  
Box b {5};  
Box c {6};  
std::vector<Box> bv {a,b,c};
```

- ▶ Unsurprisingly, changing contents of **a**, **b**, **c** does not change **bv** elements.
 - The vector **bv** has its own storage for each **Box** element.

VECTOR STORAGE OF BOX (CONT'D)

- ▶ All the other ways of iterating can act on the contents of these boxes:

```
for (Box& e : bv) {  
    std::cout << e.v << std::endl;  
}  
std::cout << std::endl;
```

```
for (Box& e : bv) {  
    e.v += 200;  
}
```

```
for (int i=0; i<bv.size(); i++) {  
    std::cout << bv[i].v << std::endl;  
}  
std::cout << std::endl;
```

```
bv[1].square();
```

```
std::vector<Box>::iterator p = bv.begin()+2;  
p->square();
```

OTHER CONTAINERS

- ▶ In addition to **`std::vector<T>`**, there is **`std::array<T>`**
 - Not dynamically resizable, maintains fixed size.
- ▶ There is a **`std::list<T>`** for linked lists.
- ▶ There are two kinds of "associative" (i.e. *key/value storage*; a *dictionary*)
 - the **`std::map<K, V>`** container is, in essence, a binary search tree.
 - ➔ It's an *ordered dictionary*.
 - the **`std::unordered_map<K, V>`** container is a hash table.
 - ➔ It's an *unordered dictionary*.

EXAMPLE USE OF `STD::MAP`

- ▶ Here is some code building a dictionary mapping strings to integers

```
#include <map>
...
std::map<std::string,int> m {};
m.insert(std::make_pair("Gwen", 49));
m.insert(std::make_pair("Carlos", 25));
m["Bob"] = 17; // also inserts
m["Gwen"] = 50; // updates
std::map<std::string, int>::iterator p = m.find("Jamie");

while (auto q = m.begin(); q != m.end(); q++) {
    std::cout << q->first << ":" << q->second << std::endl;
}
```

- ▶ The loop at the end outputs the entries in alphabetical order.

EXAMPLE USE OF `STD::MAP`

- ▶ Here is some code building a dictionary mapping strings to integers

```
#include <map>
...
std::map<std::string,int> m {};
m.insert(std::make_pair("Gwen", 49));
m.insert(std::make_pair("Carlos", 25));
m["Bob"] = 17; // also inserts
m["Gwen"] = 50; // updates
std::map<std::string, int>::iterator p = m.find("Jamie");

while (auto q = m.begin(); q != m.end(); q++) {
    std::cout << q->first << ":" << q->second << std::endl;
}
```

- ▶ The loop at the end outputs the entries in alphabetical order.
 - First **Bob : 17**. Then **Carlos : 25**. Then **Gwen : 50**.

USE OF `STD::UNORDERED_MAP` IN STATS

- ▶ The samples contain a solution of Project 1's "stats", but using STL stuff:

```
// Build a dictionary of word counts using the text entered.  
std::unordered_map<std::string,int> d { };
```

```
// Read until the end of text entry.  
while (std::cin) {  
    // Get the next line of entered text.  
    std::string line;  
    std::getline(std::cin,line);
```

```
    // Read each of the words in that line of text.  
    for ( ... /* each word w in the line */ ...) {  
        auto handle = d.find(w);  
        if (handle == d.end()) {  
            d[w] = 1; // not found; create the entry  
        } else {  
            d[w]++; // found; increment it  
        }  
    }  
}
```

USE OF `STD::UNORDERED_MAP` VECTOR ALGORITHM IN STATS

►(stats.cc continued)

```
int numWords = d.size();
std::cout << "I saw " << numWords;
std::cout << " distinct words." << std::endl;

std::vector<std::pair<std::string,int>> words { };
for (auto p = d.begin(); p != d.end(); p++) {
    words.push_back(*p);
}

auto compare = [](std::pair<std::string,int> entry1,
                  std::pair<std::string,int> entry2) -> bool
{ return entry1.second > entry2.second; };
std::sort(words.begin(), words.end(), compare);

std::cout << "The 100 most-used words are:" << std::endl;
for (int i=0; i < 99; i++) {
    std::cout << words[i].first << ":" << words[i].second << ", ";
}
std::cout << words[99].first << ":" << words[99].second << std::endl;
```

USE OF `STD::UNORDERED_MAP` `VECTOR` `ALGORITHM` IN STATS

LAMBDA

►(stats.cc continued)

```
int numWords = d.size();
std::cout << "I saw " << numWords;
std::cout << " distinct words." << std::endl;
```

```
std::vector<std::pair<std::string,int>> words { };
for (auto p = d.begin(); p != d.end(); p++) {
    words.push_back(*p);
}
```

```
auto compare = [](std::pair<std::string,int> entry1,
                  std::pair<std::string,int> entry2) -> bool
{ return entry1.second > entry2.second; };
std::sort(words.begin(), words.end(), compare);
```

```
std::cout << "The 100 most-used words are:" << std::endl;
for (int i=0; i < 99; i++) {
    std::cout << words[i].first << ":" << words[i].second << ", ";
}
std::cout << words[99].first << ":" << words[99].second << std::endl;
```

This is a lambda expression!



SUMMARY

- ▶ There are a ton of useful software components available in the C++ STL
- ▶ Many of the common data structures: sequences, stacks, queues, dictionaries
- ▶ Several useful algorithms, including sorting.
- ▶ The C++ template mechanism makes them widely applicable.
- ▶ Use them if you continue coding in C++ after this course!
- ▶ Lots of resources/tutorials on-line!
- ▶ Wednesday: lambda. Friday: smart pointers.

SUMMARY

- ▶ There are a ton of useful software components available in the C++ STL
- ▶ Many of the common data structures: sequences, stacks, queues, dictionaries
- ▶ Several useful algorithms, including sorting.
- ▶ The C++ template mechanism makes them widely applicable.
- ▶ Use them if you continue coding in C++ after this course!
- ▶ Lots of resources/tutorials on-line!
- ▶ **Wednesday: lambda.** Friday: smart pointers.

SUMMARY

- ▶ There are a ton of useful software components available in the C++ STL
- ▶ Many of the common data structures: sequences, stacks, queues, dictionaries
- ▶ Several useful algorithms, including sorting.
- ▶ The C++ template mechanism makes them widely applicable.
- ▶ Use them if you continue coding in C++ after this course!
- ▶ Lots of resources/tutorials on-line!
- ▶ **Wednesday: lambda. Friday: smart pointers.**