CS11 Intro C++

FALL 2015-2016

LECTURE 2

C++ Standard Library

- Sits on top of the C++ Core Language
  - The second fundamental component of C++
  - Extremely useful functionality!
- "Nonprimitive facilities"
  - Locale support, strings, exceptions
  - I/O streams, collections, algorithms
  - A framework for extending these facilities
- Support for some C++ language features
  - memory management, etc.

Standard Template Library

- A very well known part of Standard Library
- Primary architect: Alexander Stepanov
  - AT&T Bell Labs, then later Hewlett Packard
- Andrew Koenig motivated proposal to ANSI/ISO Committee in 1994
- Proposal accepted/standardized in 1994
- Continuous refinements, increased support
- The STL is a very sophisticated set of features…
- Fortunately, basic usage is straightforward
  - Will start learning it this week!

What Is the STL?

- A set of generic containers, algorithms, and iterators that provide many basic algorithms and data structures of computer science
- Generic
  - Very customizable; lots of templates!
- Containers
  - Collections of other objects, with various characteristics
- Algorithms
  - For manipulating the data stored in containers
- Iterators
  - "A generalization of pointers"
  - Cleanly decouple algorithms from containers

C++ Templates

- Many situations where a data structure or a function is independent of data type
  - e.g. `min()` / `max()` / `abs()` functions can work with any numeric type
  - e.g. a vector (growable array) of elements can store any element type
- C++ provides **templates** to support these cases
  - Allows a class or function to be parameterized on types and constants
- The template itself is not a class or function…
  - More like a *pattern* for classes or functions
- The template can be **instantiated** by supplying the template parameters
  - A class or function is *generated* from the template

STL Collection Templates

- The STL provides many kinds of collections
- Example: vectors – growable, resizable arrays
- Want to store a vector of integers:

```
#include <vector>
using namespace std;
...
vector<int> v;          // Vector to hold numbers
for (int i = 0; i < 10; i++) {
    int x;
    cin >> x;           // Read an integer into x
    v.push_back(x);     // Append x onto v
}
```

- Reads 10 integers from console, appends them to v

STL `std::vector`

- Example:

```
vector<int> v;  // Vector to hold numbers
```

- The template name is `std::vector`
  - Can just say `vector` since we also put "`using namespace std`" at the top
- To instantiate the template, specify the element type `int` between the angle-brackets
- The generated class' name is "`vector<int>`"
- Can use this generated class anywhere we want!
  - Local variables, function arguments, return values, data members in classes, etc.

Finding Elements

- Want to find the first element with a specific value

```
int find(vector<int> &v, int value) {
    for (unsigned int i = 0; i < v.size(); i++) {
        if (v[i] == value)
            return i;
    }
    return -1;
}
```

- Vectors can be used with array-indexing operator `[]`
  - Can assign to vector elements too: `v[i] = ...;`
- Size of vector is returned by `size()` member function
  - STL collection sizes are unsigned integers

Finding Elements (2)

- This `find()` operation is generally useful…
  - Useful for any collection type
  - Useful for any element type
- STL provides it as an **algorithm**

```
#include <algorithm>
```

- Problem: How to pass any collection type with any element type to the `find()` operation?
- Solution (part 1): Make `find()` into a function template that can also be instantiated
- Solution (part 2): Instead of passing the collection directly to `find()`, pass **iterators** to the collection

STL `std::find()` Algorithm

- Example: the `std::find()` algorithm

```
template <typename InputIterator, typename T>
InputIterator find(InputIterator a,
                   InputIterator b,
                   const T& value) {
    while (a != b && *a != value) ++a;
    return a;
}
```

Dereference to access value

Increment to move to next value

  - Searches for value in range [a, b)
- Recall: iterators are "generalizations of pointers"
- For our purposes: think of **a** and **b** as pointers
  - Can dereference an iterator to retrieve the value it points to
  - Can increment the iterator to move to the next value

STL `std::find()` Algorithm (2)

- Vectors (and all STL collections) provide two member functions that return iterators:
  - `begin()` returns iterator pointing to first element
  - `end()` returns iterator pointing just past last element
- Note: `end()` never points to an actual element in the collection!
  - Always points *just past* the last element
  - Must never dereference the `end()` iterator, or else your program will crash / mangle stuff / etc.
- To call STL `find()` algorithm on our vector:
  - `find(v.begin(), v.end(), value)`
  - Looks for `value` in entire contents of vector `v`

STL `std::find()` Algorithm (3)

- STL `find()` algorithm also returns an iterator
  - If `value` is found, returned iterator points to `value`
  - If not found, returned iterator is equal to second argument
- Example:

```
vector<int> v;
??? iter;  // TODO:  type of iter?
...
iter = find(v.begin(), v.end(), value);
if (iter != v.end())
    cout << "Found value!" << endl;
else
    cout << "Couldn't find value :(" << endl;
```

- **What is the type of `iter` ?**

STL `std::find()` Algorithm (4)

- Iterators are generalizations of pointers
  - They may be pointers
  - They may be classes that act like pointers
  - Either way, each collection may have its own special type for its iterators
- STL collections expose their iterator type through a nested typedef

```
vector<int> v;

vector<int>::iterator iter;
```

  - Use qualified name to refer to this nested typedef

STL Algorithms

- The STL has numerous useful algorithms!
- A small sampling:
  - `reverse()` reverses the contents of a collection
  - `sort()` sorts the contents of a collection (`stable_sort()` performs a stable sort)
  - `random_shuffle()` randomly shuffles the contents of a collection
  - `next_permutation()` and `prev_permutation()` generate the next and previous permutation in lexicographic order
- All these algorithms take two iterators, specifying the range of the collection to manipulate

STL Containers

- First category of containers: Sequences
  - Use indexing for access
  - Keep their values in a specific order
- `vector` – a growable array
  - Constant-time indexing, linear cost for insert/resize
- `deque` – double-ended queue
  - Constant prepend/append time, linear insert time
- `list` – a doubly linked list
  - Constant insert time, linear cost for indexing
  - Supports forward and backward traversal

More STL Containers

- The other category: Associative Containers
  - Use keys for access – unique values, any type
  - Tend to keep values in a specific order, but not required to!

- `set`, `multiset` – ordered collection of keys
  - In set, keys are unique; in multiset, can appear multiple times
  - log(n) time for insert, lookup
- `map`, `multimap` – ordered collection of (key, value) pairs
  - In map, keys are unique; in multimap, can appear multiple times
  - log(n) time for insert, lookup
  - Entries are ordered by key

Even More STL Containers

- Unordered sets and maps hash their keys instead of sorting them
- `unordered_set`, `unordered_multiset` – unordered collection of keys
  - Like `set` and `multiset`, but with constant insert/lookup time
- `unordered_map`, `unordered_multimap` – unordered (key, value) pairs
  - Like `map` and `multimap`, but with constant insert/lookup time

The Standard Template Library

- Much more to discuss with the Standard Template Library…
  - Will continue discussing its nuances all the way into the Advanced C++ track
- A very useful tool – particularly on homework assignments ☺

Unintended Side Effects…

- Want to pass objects to functions by-reference…
  - Much faster, when objects are expensive to copy
  - e.g. an STL vector of 100 3D points!
- A function can mutate its arguments…
- Can easily have situations where a function *accidentally* mutates an argument
  - Unintended side-effects
- These bugs are very difficult to identify!
  - Time between the actual bug occurring, and the manifestation of the defect, can be very large

Preventing Side Effects

- Can prevent changes to a variable by declaring it **const**
- Example: compute sum of a vector of floats

```
float sum(const vector<float> &v) {
    float sum = 0;
    for (unsigned int i = 0; i < v.size(); i++)
        sum += v[i];
    return sum;
}
```

- The function may not change v, only read from it
  - If function tries to write to v, the program will not compile
- **When you pass objects by reference, and you want to avoid side-effects, make the reference const!**

Preventing Side Effects (2)

- Using our `sum()` function:

```
vector<float> values;
for (int i = 0; i < 20; i++) {
    float f;
    cin << f;
    values.push_back(f);
}
cout << "Sum = " << sum(values) << endl;
```

- `const` only applies inside the `sum()` function
  - (It doesn't make `values` read-only for everyone!)

sum() with Iterators

- Can rewrite our `sum()` function with iterators

```
float sum(const vector<float> &v) {
    float sum = 0;
    vector<float>::iterator it = v.begin();
    while (it != v.end()) {
        sum += *it; // Read current value
        it++;       // Go to next value
    }
    return sum;
}
```

- **This version won't compile ☹**
- Problem: the iterator allows changes to `v`, so the compiler can't allow you to retrieve it

sum() with Iterators (2)

- Instead, use vector<float>::const_iterator

```
float sum(const vector<float> &v) {
    float sum = 0;
    vector<float>::const_iterator it = v.begin();
    while (it != v.end()) {
        sum += *it; // Read current value
        it++;       // Go to next value
    }
    return sum;
}
```

- const_iterator is just another version of the iterator that disallows changing the collection

`const` Member Functions

- Member functions that don't change the object they are called on, can be marked `const`

```
class Point {
    ...
public:
    float getX() const;
    float getY() const;
};
```

  - `getX()` and `getY()` don't change the object they are called on
  - (Obviously, can't mark `setX()` or `setY()` as `const`...)
- `const` is part of the member function's signature
  - Must specify `const` in both declaration and definition

`const` Member Functions (2)

- Once `Point` member functions are properly marked `const`, can use `Point`s in `const` vectors

```
void printPoints(const vector<Point> &pts)
```

- Only the member functions of `Point` that are marked `const` may be called through `pts`
  - `cout << pts[i].getX() << endl;  // OK`
  - `pts[i].setX(45);  // Error`

This Week's Assignment

- Traveling Salesman Problem
- Given a set of points, choose an order to visit all of the points that minimizes the distance traveled
  - Must start and end at the same point
- Easy to write a brute-force solution using the STL
  - Use `std::vector` and some STL algorithms
  - It will rapidly slow down as the number of points increases…
- Use your `Point` class from last week to represent the points!