CS11 – Advanced C++

Winter 2014-2015

Lecture 1

Welcome!

- 7-9 lectures
  - Slides are posted on CS11 website
  - http://courses.cms.caltech.edu/cs11
- 7 lab assignments
  - Advanced tracks tend to focus on a larger project
  - This term’s project: write a simple ray tracer
  - A lot of code to write, but a lot of fun to play with

Lab Submissions

- Advanced C++ track *requires* a CS cluster account
- Using csman homework submission website:
  - http://csman.cms.caltech.edu
  - Uses CS cluster for authentication
- Will also be using Git this term
  - Can host your repository on the CS cluster, or elsewhere if you prefer

The 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 language features
  - memory management, runtime type information (RTTI)
- <u>Portable</u> implementations of useful functions
  - **`sqrt()`**, **`memmove()`**, etc. (not optimized!)

Standard Template Library (STL)

- 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

What *Is* the STL?

- A set of generic containers, algorithms, and iterators that provide many of the basic algorithms and data structures of computer science.
- Generic
  - <u>Heavily</u> parameterized; 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.

STL Underlying Concepts

- Working with STL requires fluency with:
  - Class inheritance
  - Class templates and function templates
  - Function pointers
  - Basic data structures/computational complexity
- Some of these concepts may be a little new
  - Don’t worry; we will explore them all!

Simple STL Example!

- You want an array of numbers.

```
vector<int> v(3);  // Vector of 3 elements
v[0] = 7;
v[1] = v[0] + 3;
v[2] = v[0] + v[1];
```

- Now you want to reverse their order!

```
reverse(v.begin(), v.end());
```

- **`vector<int>`** is the generic container
- **`reverse()`** is a generic algorithm
- **`reverse()`** uses iterators associated with **`v`**

STL Algorithms

- Generic function-templates
  - Parameterized on iterator type – *not* container
- Example: the **`find()`** algorithm

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

  - Searches for **`value`** in range **`[first, last)`**.

Algorithms and Iterators

```
template <typename InputIterator, typename T>
InputIterator find(InputIterator first,
                   InputIterator last,
                   const T& value) {
```

- **`InputIterator`** isn't a specific type!

  ```
  while (first != last && *first != value) ++first;
  ```

  - Just needs to support **`*`** (dereference), **`++`** (increment), and equality operators.
- Pointers also satisfy these constraints.

```
float a[5] = { 1.1, 2.3, -4.7, 3.6, 5.2 };
float *pVal;
pVal = find(a, a + 5, 3.6);  // float* as iterators
```

The Big Picture

- This set of required functionality for the iterator-type is called a <u>concept</u>.
  - In this case, the concept is named “InputIterator.”
- A type that satisfies these requirements is said to “model the concept.”
  - Or, it “conforms to the concept.”
- For example:
  - “**`int*`** is a model of Input Iterator because **`int*`** provides all of the operations that are specified by the Input Iterator requirements.”

What about `reverse()`?

- The `reverse()` algorithm needs more!
  - Specifically, its iterators <u>also</u> need `--` operator.
- `reverse()`'s arguments must model the <u>BidirectionalIterator</u> concept.
  - Like InputIterator, but with more requirements.
- BidirectionalIterator <u>refines</u> the InputIterator concept.
  - This is *exactly like* class-inheritance.
  - Different terms because these *aren't* classes.

Wait A Minute…

- A major issue with this whole “concept” thing:
  - *No language support whatsoever* for declaring or enforcing the requirements of concepts!
  - *No language support* for declaring that a particular type models a concept.

- This makes it a bit challenging.

Iterator Concept Hierarchy

- Trivial Iterator – supports dereference
  - That’s it. Yep, it’s trivial.
- Input Iterator – supports increment
  - Only read support is guaranteed.
  - Only single-pass support guaranteed.
- Forward Iterator – like Input Iterator
  - Supports multi-pass algorithms.
- Bidirectional Iterator – supports decrement
- Random Access Iterator
  - Supports arbitrary-size steps forward and backward

Output Iterators

- Output Iterators don’t appear in the iterator concept hierarchy
- Different, very limited set of requirements
  - Support assignment
  - Support increment
  - Support postincrement-and-assign
    - **`*iter++ = value;`**
- “It’s like a tape.”
  - You can write to the current location
  - You can advance to the next location

Function Objects

- Anything that can be called like a function
  - A generalization of functions
  - Can be a true function pointer
  - Can be an instance of a class that overloads `()`
- Allows customization of algorithm operations
  - Can pass these things to STL algorithms
- Also known as “functors”

Function Pointers?!

- C/C++ functions can be referred to by name
  - **`sin(x)`**, **`cos(x)`**, **`sqrt(x)`**, etc.
- Can also refer to functions via <u>function pointers</u>
  - Like a normal pointer, but function can be called through it
  - Function's signature is part of the pointer's type
    - Number and types of arguments, return type
- Above funcs take a **`double`** and return a **`double`**
  - A function pointer for them could be like this:

    ```
    double (*fp)(double);
    ```
  - Variable name is **`fp`**
  - Points to a function that takes a **`double`** and returns a **`double`**

Using Function Pointers

- Normally refer to functions to invoke them

```
double rot = coord * sin(angle);
```

  - Invokes **`sin`**, using **`angle`** as argument
- Can also get a function's address via its name

```
double (*fp)(double);
...
fp = sin;  // No arguments to sin here!
...
double res = fp(input);
```

  - Use **`fp`** like a normal function
  - Can set **`fp`** to *any* function with the same signature
    - **`sin`**, **`cos`**, **`tan`**, **`sqrt`**, **`log`**, **`exp`**, your own functions, etc.

Functor Concepts

- Generator `f()`
  - No arguments.
- Unary Function `f(x)`
  - One argument.
- Binary Function `f(x, y)`
  - Two arguments.
- Special concepts for `bool` return-types
  - Predicate `bool p(x)`
  - Binary Predicate `bool p(x, y)`
- Others, too…

Simple Functor Example

- You want a collection of 100 random values

```
vector<int> values(100);
generate(values.begin(), values.end(), rand);
```

- Can create your own functions

```
int randomColorValue() {
  return rand() & 0x00FFFFFF;
}
...
vector<int> randColors(10);
generate(randColors.begin(), randColors.end(),
  randomColorValue);
```

Functors with State

- You want to find the sum of those values
  - Need a functor with state
  - A class with overloaded `()` is perfect for this

```
struct adder : public unary_function<int, void> {
  int sum;
  adder() : sum(0) { }
  void operator()(int x) { sum += x; }
};
```

Argument Type Result Type

- Apply functor with **`for_each`** algorithm

```
adder result =
  for_each(values.begin(), values.end(), adder());
cout << "Sum is " << result.sum << endl;
```

The `for_each()` Algorithm

- Example implementation of `for_each()`:

```
template <typename InputIterator, typename Function>
Function for_each(InputIterator first,
                  InputIterator last, Function f) {
  while (first != last) {
    f(*first);
    ++first;
  }
  return f;
}
```

- Our example:

```
adder result =
  for_each(values.begin(), values.end(), adder());
```

  - An `adder` object is initialized, and a *copy* is passed to `for_each()`
  - Function-template uses object `f` as a function on each element
  - Function returns the object `f`, which is then copied into `result`

Printing The Numbers

- Now you want to print the numbers, separated with commas.
- Use **`copy()`** algorithm and Output Iterators

```
copy(values.begin(), values.end(),
  ostream_iterator<int>(cout, ", "));
```

  - Note that **`ostream_iterator`** template-param must match element-type of collection.

STL Containers

- Sequences are a refinement of Container concept
  - Elements arranged in linear sequential order
  - Variable size; can grow or shrink
- **`vector`** – random access, constant append time, linear insert time, linear prepend time
- **`deque`** – like **`vector`**, but constant prepend time too
- **`list`** – doubly linked list, constant insert anywhere, only sequential access
- **`slist`** – singly linked list, only forward traversal
- **`bit_vector`** – vector of **`bool`**s, optimized for space!

Associative Containers

- Support efficient retrieval based on keys
- No support for inserting at a specific position
- **`set`** – stores keys; each appears only once
- **`map`** – stores (key,value) pairs; each key appears only once
- **`multiset`**, **`multimap`** – like the above, but keys can appear multiple times
- These are Sorted Associative Containers
  - They don't hash the keys! Most operations are O(log(N))
  - But, they do keep their entries sorted by key.

Extensions to STL Containers

- Hashed Associative Containers have constant-time insert/retrieve operations
- Are considered "STL Extensions"
- **`hash_set`**, **`hash_map`** – like **`set`**, **`map`**
- **`hash_multiset`**, **`hash_multimap`** – like **`multiset`**, **`multimap`**
- Unlike the other Associative Containers, keys aren't kept in a specific order.

Container Adaptors

- "Provides a restricted subset of functionality"
- Uses another container for internal storage
- **`stack`** – LIFO, uses **`deque`** by default
- **`queue`** – FIFO, also uses **`deque`** by default
- **`priority_queue`** – uses **`vector`** by default

- Can override the default internal container

Containers and Iterators

- STL containers provide iterators over their elements
  - `begin()` returns an iterator to the first element
  - `end()` returns an iterator “just past” the last element
- Type of “element” depends on the container!
  - Sequences are simple – element type is what’s specified in template parameter
    - `vector<int>` has elements of type `int`
  - Associative containers contain (key, value) pairs
    - `map<string, int>` has elements of type `std::pair<string, int>`
- “Element type” is called the container’s <u>value type</u>

Iterator Types

- STL containers provide definitions of value type, iterator types as nested typedefs

```
vector<int> values(100);
...
vector<int>::iterator iter = values.begin();
while (iter != values.end()) {
  ... // Compute stuff.
}
```

- Sometimes you need to do this…
  - Prefer to use provided algorithms instead, unless it's just too complicated or annoying.

Helpful Resources

- The SGI STL Documentation
  - http://www.sgi.com/tech/stl/index.html

VERY USEFUL!

- Effective STL by Scott Myers