Advanced C++

March 7, 2013

Mike Spertus

mike_spertus@symantec.com

`YIM: spertus`

A TOUR OF THE STANDARD LIBRARIES

Ones we won’t discuss

- Ones we’ve already covered
  - <iostream> and related
  - <locale>
  - <thread>
  - <atomic>
  - <string>
- Ones we’ll cover next quarter
  - <type_traits>
  - <iterator>

Standard exception types

- Even though technically, you can throw exceptions of any type, you should always have your exceptions inherit from std::exception, std::runtime_error, or std::logic_error
- Another good best practice: Throw by value but catch by reference
- Remember, don’t use exception specifications
  - But note that C++11 introduces noexcept keyword (beyond the scope of this quarter, but will come back to it next quarter)

Algorithms

- `for_each`
- `find`
- `find_if, find_if_not`
- `find_first_of`
- `adjacent_find`
- `count, count_if`
- `mismatch, equal`
- `is_permutation`
- `search, search_n, binary_search`

More algorithms

- `copy,copy_n,copy_if,copy_backward`
- `move,move_backward`
- `iter_swap`
- `transform`
- `replace,replace_if`
- `Generate`
- `rotate,rotate_copy,random_shuffle,shuffle`
- `all_of,any_of,none_of`
  - Check if all/any/none of the items in a container (or range) have a certain property
  - Creating an example will be part of your job in the HW

Follow remove with erase

- This is item 32 of Effective STL
- Otherwise you won’t get rid of anything!
- To take all of the 99s out of a vector:

```
v.erase(remove(v.begin(), v.end(), 99),
          v.end());
```

More algorithms

- `copy_n`

```
vector<int> v = getData();
// Print 5 elements
copy_n
  (v.begin(), 5,
   ostream_iterator<int>
    (cout, "\n"));
```

- Bet you’ve wished this was in C++ for years

More algorithms

- find_if_not

```
vector<int> v = { 1, 3, 5, 6, 7};

// Print first elt that is not odd
cout << *find_if_not
           (v.begin(),
            v.end(),
            [](int i) {
              return i%2 == 1;
            });
```

More algorithms

- partition_copy

```
vector<int> primes;
vector<int> composites;
vector<int> data = getData();
extern bool is_prime(int i);

partition_copy
  (data.begin(),
   data.end(),
   back_inserter(primes),
   back_inserter(composites),
   is_prime);
```

More algorithms

- `minmax,minmax_element`
  - Gets both the biggest and smallest items in the range
- `Sort variants`
  - `sort, stable_sort, partial_sort, nth_element, merge`
- `is_heap, is_heap_until, is_sorted, is_sorted_until, partial_copy`
- Set operations
  - `include,set_union, set_intersection, set_difference, set_symmetric_difference`

Emplace

- Rather than incur the overhead of copying/moving a new object into a container, you can just construct it in place
-
```
vector<thread> threads;
// How we did it in cspp51044
threads.push_back(thread(f, 7));
// How we can do it with emplace
threads.emplace_back(f, 7);
```

Functors

- Recall that a functor is “anything that is callable”

-
  ```
  struct A {
    char operator()() {
      return 'A';
    }
  };
  ```

-
  ```
  A a;
  // a can be called like a function
  cout << a() << endl; // Prints A
  ```

The problem with function pointers

- We can declare a pointer to a function like

```
int(*fp)(int, double);
```

- However, you can't assign a functor to `fp`

```
int f(int, double);
struct Functor {
   int operator()(int, double);
};
fp = &f; // OK
Functor fun;
fp = &fun; // Error: not a function
```

std::function to the rescue

- C++11 has a function class that can also take anything that is callable with the right arguments (On old compilers, you can use boost::function)

```
#include<functional>
function<int(int, double)> fp;
fp = &f; // OK
fp = &fun; // OK
```

Using `std::bind`

- Bind lets you reduce or reorder the number of arguments in a function.
- For example, suppose f is declared as

  ```
      int f(int, int)
  ```

  Then

  ```
      bind(f, 3, _1)
  ```

  is functor of one variable that calls `f`:

  ```
      bind(f, 3, _1)(7)
  ```

  is the same as

  ```
      f(3, 7)
  ```

- Similarly, `bind(f, _2, _1)(x, y)` is the same as `f(y, x)`
- Placeholders can be repeated:

  ```
      bind(std::multiplies<int>, _1, _1)(7) == 49
  ```

- Used to be a very common way of creating functors, but in C++11, you can usually use lambdas instead.

More function and bind examples

- They are now part of the C++11 standard libraries

-
  ```
  function<int(int,int)>
    times = std::multiplies<int>;
  ```

-
  ```
  function<int(int)>
    square
      = bind(std::multiplies<int>,
                  _1, _1);
  ```

Containers

- Sequences
  - vector, array, deque, list, forward_list, queue, priority_queue, bitset (don’t use vector<bool>) and stack
- Associative containers
  - set, unordered_set, map, unordered_map
- heap
  - Maybe next quarter

Singly linked lists

- `std::list` is a doubly-linked list
- Now there is a singly-linked list `std::forward_list`
- Interestingly, it has no `size()` method because calculating the size of a linked list is expensive.
  - See http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2543.htm for a discussion of design decisions

Hash tables

- There was wide desire to add hash tables to C++11
  - `std::map` requires that its elements a “less than comparable,” but there is not always a natural ordering
  - `std::map` may be much slower than a true hashtable on large collections
- Google code search (now defunct ☹) showed that we couldn’t call them hash_table.

Unordered maps

- std::unordered_map acts more or less just like a std::map
  - Instead of std::less, it uses std::hash by default
  - Of course, you can specify your own hash function
- If you iterate the elements of std::map, you get them in order, but for a std::unordered_map, you don't get them in any particular order
- There are also unordered_set, unordered_multimap, and unordered_multiset

Making stack exception-safe

- You would expect to be able to pop an object of a stack

  -
    ```
    stack<A> stk;
    ...
    A a(stk.pop()); // Illegal!
    ```
- The problem with this would be if A’s copy constructor threw an exception.
  - The top element could be lost forever
- Instead, stack::pop has void return type.
- Do the following instead

  -
    ```
    stack<A> stk;
    ...
    A a(stk.top());
    stk.pop();
    ```

Random number generators

- Can specify a distribution function
  - E.g., uniform_int_distribution or normal_distribution
- Very powerful, but a little tricky to use
- Let’s look at the example from Bjarne Stroustrup’s FAQ

Throwing dice

```
// distribution that maps to the ints 1..6
uniform_int_distribution<int> one_to_six {1,6};
default_random_engine re {}; // the default engine
```

To get a random number, you call a distribution with an engine:

```
int x = one_to_six(re); // x becomes a value in [1:6]
```

Passing the engine in every call can be considered tedious, so we could bind that argument to get a function object that we can call without arguments:

```
auto dice {bind(one_to_six,re)}; // make a generator
int x = dice(); // roll the dice
```

Regular expressions

- C++11 adds pattern matching
- Just like string is really a typedef for basic_string<char>, regex is a typedef for basic_regex<char>, so different character types can be handled.

regex_match

-
  ```
  string text("How now, brown cow");
  cout << std::alphabool;
  regex ow("ow");
  regex Hstarw("H.*w");
  // False
  cout << regex_match(text, ow);
  // True
  cout << regex_match(text, Hstarw);
  ```

regex_search

- Can just check for matching substring
```
// True
cout << regex_search(text, ow);
```
- Or can find all matches
```
cmatch results;
regex_search(text, results, Hstarw);
// Prints "How"
copy
  (results.begin(),
   results.end(),
   ostream_iterator<string>(cout, "\n"));
```

Smart pointers

- shared_ptr and unique_ptr are smart pointers
  - A unique_ptr is the unique owner of an object
    - unique_ptr is movable, allowing easy transfer of ownership
  - A shared_ptr shares ownership of an object
    - When the number of owners of a shared object go down to zero, the object is deleted
    - Reference counting
- Remember, auto_ptr is deprecated in favor of unique_ptr

Improved time facilities

- See http://www2.research.att.com/~bs/C++0xFAQ.html#std-duration

Tuples

- Tuples are a generalization of std::pair to any number of fields

```
tuple<string, int> si("str", 2);
// di will be a tuple<double,int, char>
auto di = make_tuple(2.5, 3, 'c');
cout << get<0>(di) // prints 2.5
int three = get<1>(di);
```

Tuples

- Tuples are very useful for creating compound types on the fly
- We will implement an improved version of tuple from scratch in a few weeks

HW 9-1

- Use all std::all_of to check if all of the numbers in the vector (2, 4, 7, 542, 211, 6) are prime

HW 9-2

- Write a little code to use the following
  - New algorithms
    - all_of, any_of, none_of, copy_n, find_if_not, partition_copy, minmax, minmax_element
    - Extra credit: is_heap, is_heap_until, is_sorted, is_sorted_until, partial_copy
  - New containers
    - forward_list, unordered_map
    - Does your standard library implementation support emplace?
    - Extra credit: unordered_set, unordered_multimap, unordered_multiset
  - Random (generate a normal distribution)
  - Shared_ptr and unique_ptr (negative points for using auto_ptr)
  - Regex
  - Tuple
  - Extra credit: time

HW 9-3: Extra credit

- Create a hash table of strings that uses md5 as its hash function

HW 9-4: Extra Credit

- Give an example of where you might prefer using bind to lambdas