Iterators and STL Sequence Containers

Topics Covered

- Reading: 810-835, 997-1027 [Appendix G]
- The iterator concept
- The iterator hierarchy
  - Input, output, forward, bidirectional and random-access
- Stream iterators & insertion iterators
- Container methods
- Sequence containers & sequence adapters
- Homework

Why Iterators? I

- To make algorithms independent of container
  - Consider finding a value in an array
    ```
    int * find(int *a, int n, int val) {
      for (int i = 0; i < n; i++)
        if (val == a[i]) return &a[i];
      return 0;
    }
    ```
  - Templates let us use any type instead of int
  - But algorithm is still tied to array container
Why Iterators? II

- Consider code for find using a linked list
  - Assume struct node : next, data
    ```c
    int * find(node* head, int val) {
        node * p = head;
        for(; p != 0; p = p->next)
            if (p->data == val)
                return &(p->data);
        return 0;
    }
    ```

Why Iterators? III

- Both methods use same "general" algorithm
  - 1.) Traverse entire container
  - 2.) Look for value in each position
  - 3.) Return reference (or pointer) to value if found
  - 4.) Return 0 if not found

- Template generalizes type of data
- Iterator generalizes access and movement
  - Iterator is a generalization of the pointer concept

Why Iterators? IV

- Can use classes to create such a "logical pointer"
  - The iterator class needs:
    - 1.) Dereferencing [*itr ]
    - 2.) Assignment [ itr1 = itr2 ]
    - 3.) Comparison [ itr1 == itr2, itr1 != itr2 ]
    - 4.) Traversal [ ++itr, itr++ ]

- Each container creates its own iterator class
  - User does not care how operations defined
Kinds of Iterators

- The STL defines an iterator "hierarchy"
  - One-way, one-pass iterators
    - Input iterator: read values in a container
    - Output iterator: write values to a container
  - Forward iterator: one-way, strictly ordered
  - Bidirectional iterator: forward & backward
  - Random access iterator: pointer addition
- "Higher" iterators include "lower: capabilities

Input Iterators I

- Consider a generic find() method
  ```cpp
  template <class Iter, class T>
  Iter find(Iter first, Iter last, const T& value) {
    while (first != last && *first != value) {
      ++first;
    }
    return first;
  }
  ```
- Iter must support first != last, *first, first++
- No need to go backwards or re-read any value
- These are the requirements of an input iterator

Input Iterators II

- Input iterators are not a type, but a category
  - Anything that meets these requirements qualifies
    - Equality comparison: == and !=
    - Increment: ++ (prefix and postfix)
    - Dereferencing: *
  - An input iterator only goes forward, and only once
  - Things that an input iterator needn't do
    - "Pointer" arithmetic: +, -, *++, -=, [ ]
    - Assignment: iterators need not be saveable
Input Iterators III

You can use find() with any iterator that qualifies
  int* ptr = find(a[0], a[5], 7); // first 7
- list<int> lst; lst(a[0], a[5]);
  list<int>::iterator i;
  i = find(lst.begin(), lst.end(), 7);
- istream_iterator<char> in(cin);
  istream_iterator<char> eos;
  find(in, eos, 'x');

- Regular pointer, list iterator, input stream iterator

Output Iterators I

Output iterators can write; need not read
- Just like you can write to cout, but not read from it
- Guarantees *itr = ..., but not val = *itr;
- Other requirements are same as input iterator

Where would you use an output iterator?
- Used for generic algorithm like copy:

  template <class InItr, class OutItr>
  OutItr copy(InItr first, InItr last, OutItr result) ...

Output Iterators II

First two arguments are InItr, last is OutItr
- Any iterator that meets requirements can be used
- Regular array can both read and write

  int a[5] = {1, 2, 3, 4, 5};
  b[5];
  copy(a, a+5, b);

- An ostream_iterator can only write:

  list<int> lst(b, b+5);
  ostream_iterator<int> out(cout, " ");
  copy(b, b+5, out);
Forward Iterators I

- Forward iterators are one step higher
  - Can read like input-iterators
  - Can write like output-iterators
  - Can only move forward like input/output iterators
- Main difference? Forward iterators are saveable
  - That means that multi-pass algorithms are possible
    » Calculating the standard deviation, for instance
  - The container must guarantee order is stable

Forward Iterators II

- Here's an algorithm that requires a forward iterator
  ```cpp
  template <class ForwardItr, class T>
  void replace(ForwardItr first, ForwardItr last,
               const T& x, const T& y)
  {
    while (first != last) {
      if (*first == x)         // read
        *first = y;           // write
      ++first;
    }
  }
  ```
  - Note that iterator must both read and write

Bidirectional Iterators

- Bidirectional iterator supports forward and reverse traversal of a container
  - Means it must support both versions of --
    » Otherwise, same requirements as a forward
  - reverse() algorithm requires bidirectional iterators
    ```cpp
    int a[5] = { 1, 2, 3, 4, 5 };
    reverse(a, a+5);
    list<int> lst(a, a+5);
    reverse(lst.begin(), lst.end());
    ```
Random Access Iterators

- Random access iterators add these requirements
  - Addition and subtraction: \(i + n, n + i, i - n\)
  - Subscript operator: \([i][n]\)
  - "Big Jumps": \(i += n, i -= n\)
  - Iterator subtraction: \(i1 - i2\) (result is integer)
  - Comparisons: \(i1 > i2, <, <=, >=\)

- **Needed for functions like sort:** iterator05.cpp
  - Others, like binary_search, just perform better

Reverse Iterators I

- How do you move backwards through a list?
  - This doesn't quite work [begin is in, end is out]:
    ```cpp
    for (i = v.end(); i != v.begin(); i--)
        cout << *i << endl;
    ```
  - Solution? a reverse_iterator [reverse_iter01]
    - Moves backwards when incremented
    - Container must support random-access or bidirectional iterators

Reverse Iterators II

- Reverse iterators are produced by:
  - rbegin(): last element in container
  - rend(): element before first in container
  - Use a reverse iterator like this:
    ```cpp
    vector<int> vec;
    for (int i = 0; i < 5; i++)
        vec.push_back(rand());
    vector<int>::reverse_iterator ri;
    for (ri = v.rbegin(); ri != v.rend(); ri++)
        cout << *ri << " ";
    ```
Stream Iterators I

Stream iterators allow you to treat input and output as another form of "container"
- Can use these iterators with generic algorithms
- Consider the copy(i1, i2, i3) algorithm
  » Copies the range [i1, i2) to range starting at i3
  » i3 must be [at least] an output iterator
- An ostream_iterator lets you turn cout into such an output iterator

Stream Iterators II

The ostream_iterator has two constructors:

- ostream_iterator<T>(ostream&);
- ostream_iterator<T>(ostream&, const char * sep);
- Notice that the iterator must know what T is output
- Print chars with no space and integers with space

- ostream_iterator<char> char_out(cout);
- ostream_iterator<int> int_out(cout, " ");

Stream Iterators III

Writing to ostream_iterator puts value in stream
- Write via a generic algorithm like copy()
  - copy(a1, a1 + 5, char_out);
- Write via dereferencing operator
  - for (int i = 0; i < 5; ++i) "int_out = a2[i];"
- The following is just a no-op
  - int_out++;
Stream Iterators IV

- `istream_iterators` do a similar trick with `cin`, etc.
  - Here are the constructors
    - `istream_iterator<T>();` // "end of input"
    - `istream_iterator<T>(istream&);` // reads from
  - Can only read type of T (like `ostream_iterator`)
- Use with generic functions or as input iterator
  - `merge(v1.begin(), vi.end(), int_in, int_end, out);`

Insert Iterators I

- Output algorithms overwrite output range
  - Consider this `copy()` command:
    - `copy(v1.begin(), v1.end(), v1.begin());`
  - `v2` must be large enough to receive range
  - Previous contents of output are overwritten
  - What if output range is smaller than `v2`?
    - A runtime error occurs

Insert Iterators II

- The `back_inserter` adds an item to the end of a sequence, and, if necessary, expands it
  - Use the `back_inserter()` generic function
    - Pass the container as the argument
  - Get back an iterator that uses `push_back()` instead of `* dereferencing operator`
    - `copy(v1.begin(), v1.end(), back_inserter(v2));`
Insert Iterators III

- The front_inserter works the same way, but uses `push_front()` instead of `push_back()`
  - Only used with containers having a `push_front` method
    ```cpp
    copy(i1, i2, front_inserter(lst2));
    ```
  - The inserter iterator uses `insert()` and works for all sequences. Pass it a container and a regular iterator:
    ```cpp
    copy(i1, i2, inserter(v, v.begin() + 2));
    ```

STL Container Concepts I

- What is an STL container?
  - An object that stores other objects
    - All objects must be of the same type
    - Data stored in a container is "owned" by container
    - Expires when container expires
  - Objects stored in a container must be
    - Assignable and copy-constructable
- No order assumed by container

STL Container Concepts II

- STL has several container refinements
  - Sequence [linear] containers
    - A first and last element
    - Remaining elements have one next, previous
    - Sequence models: vector, list, deque
    - Adapters : queue, stack, priority_queue
  - Associative containers
    - Nonlinear, organized with key-value pairs
    - Models : set, multiset, map, multimap
Container Methods

- All containers have these methods [container01]
  - X::iterator i; [Create iterator type for container]
  - X a; [Default constructor. 0 size]
  - X a(b); [Copy constructor]
  - X a = b; [Same effect]
  - a.begin(), a.end() [First, one-past-last iterators]
  - a.size(); [Num of elements container]
  - a.swap(b); [Exchange contents]
  - a == b, a != b [Size and contents same]

Sequence Container Methods

- All sequence containers have these: [sequence01]
  - X a(n, t); // Construct with n copies of t
  - X b(i1, i2); // Construct with iterator range [i1, i2]
  - a.insert(i, t); // Insert t before i
  - a.insert(i, n, t); // Insert n ts before i
  - a.insert(i, i1, i2); // Insert range [i1, i2]
  - a.erase(i); a.erase(i1, i2); // Erase elements
  - a.clear(); // Remove all elements

Optional Sequence Methods I

- Operation "expense" depends on container
- Constant time operations
  - Don't depend upon size of container
    - Insert a new node in a linked-list
    - Add a new value to the end of an array
- Linear time operations
  - Time is proportional to number of elements
    - Retrieving the nth element in a linked list
    - Inserting at the beginning of an array
Optional Sequence Methods II

- Accessing container elements
  - All sequence containers support `front()`, `back()`
    - Returns a reference to first and last element
      - `vec.front() = 13;`
      - `cout << deq.back();`
  - Only vector and deque support random access
    - Use subscripting [no error checking]
    - Use the `at()` method, catch `out_of_range`

Optional Sequence Methods III

- Add elements to ends of container
  - Use `push_front()` and `push_back()`
    - The vector does not support `push_front()`
  - Methods increase size of container
- Remove elements from ends of container
  - Use `pop_front()` and `pop_back()`
    - The vector doesn't support `pop_front()`
    - These methods remove, don't return value

The list Class

- Implements a doubly-linked list of `T`
  - Fast insertion and deletion of any element
    - Iterators remain valid after insert/delete
    - Compare to vector and deque [linear-time]
  - No random access
    - Have to "walk" list to find element [linear time]
    - Supports all sequence methods but `[]` and `at()`
- Example list01.cpp [note many member functions]
The deque Class

- A double-ended queue [pronounced “deck”]
  - Allows random access like vector
  - Allows fast insert at either end like a list
  - Not stored contiguously like vector
    » Member function capacity() is not defined
    » Stored in non-contiguous subsegments
    » More complex code; is less efficient than vector
  - Example: deque01.cpp

Sequence Adapters I

- What are sequence adapters?
  - Classes that provide a new interface for an underlying container such as list and deque
- Built-in adapters
  - stack using vector, list, or deque
    » stack<int> istack;
    » stack<float, list<float>> fstack;
  - queue using list or deque
  - priority_queue using vector or deque

Sequence Adapters II

- The stack class [last-in, first-out] [stack01.cpp]
  - More restrictive than underlying container
    - bool empty(); // Any items?
    - int size(); // Num items
    - T& top(); // Retrieve first item
    - void push(const T&); // Add item
    - void pop(); // Remove first item
  - Note that pop() only removes, doesn't retrieve
    » Use: val = s.top(); s.pop();
**Sequence Adapters III**

- The queue class [first-in, first-out] ([queue01.cpp])
  - Used to buffer production and consumption
    - Examples: a print queue, receiving network packets
  - Cannot use a vector as underlying container
    - The vector can only insert and remove at 1 end
  - Methods
    - size(), empty(), front(), back()
    - push() // add item to back of queue
    - pop() // remove item at front of queue

**Sequence Adapters IV**

- What is a priority_queue? ([priority_queue01.cpp])
  - A "sorted" queue
  - Most "important" item is always at the front
    - Example: a CPU scheduler with priority
  - Change meaning of "important" using function
    ```cpp
    priority_queue<int> pq1;
    priority_queue<int> pq2(greater<int>);
    ```
  - Same functions as stack class [not queue]

**Homework**

- Due Sunday, May 16
- Homework 26: Add random access to list
  - Derive a new class, raList, from the list class
  - Add constructors, const, non-const subscript operators
  - Use hmwk26.cpp to test, submit only raList.h file
- Homework 27: Frequency Counter II
  - Re-do Homework 25 using the list and pair classes
  - Tell me which one was easier