Learning Objectives

• Iterators
  – Constant and mutable iterators
  – Reverse iterators

• Containers
  – Sequential containers
  – Container adapters stack and queue
  – Associative Containers set and map

• Generic Algorithms
  – Big-O notation
  – Sequence, set, and sorting algorithms
Introduction

• Recall stack and queue data structures
  – We created our own
  – Large collection of standard data structures exists
  – Make sense to have standard portable implementations of them!

• Standard Template Library (STL)
  – Includes libraries for all such data structures
    • Like container classes: stacks and queues
Iterators

• Recall: generalization of a pointer
  – Typically even implemented with pointer!

• "Abstraction" of iterators
  – Designed to hide details of implementation
  – Provide uniform interface across different container classes

• Each container class has "own" iterator type
  – Similar to how each data type has own pointer type
Manipulating Iterators

• Recall using overloaded operators:
  – ++, --, ==, !=
  – *
    • So if p is an iterator variable, *p gives access to data pointed to by p

• Vector template class
  – Has all above overloads
  – Also has members begin() and end()
    c.begin(); //Returns iterator for 1st item in c
    c.end(); //Returns "test" value for end
Cycling with Iterators

• Recall cycling ability:
  for (p=c.begin(); p!=c.end(); p++)
    process *p    // *p is current data item

• Big picture so far...

• Keep in mind:
  – Each container type in STL has own iterator types
    • Even though they’re all used similarly
//Program to demonstrate STL iterators.
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;

int main( )
{
    vector<int> container;

    for (int i = 1; i <= 4; i++)
        container.push_back(i);

    cout << "Here is what is in the container:
";
    vector<int>::iterator p;
    for (p = container.begin( ); p != container.end( ); p++)
        cout << *p << " ";
    cout << endl;

    cout << "Setting entries to 0:
";
    for (p = container.begin( ); p != container.end( ); p++)
        *p = 0;
}
cout << "Container now contains:\n";
for (p = container.begin(); p !=
     container.end(); p++)
    cout << *p << " ";
cout << endl;
return 0;
}

SAMPLE DIALOGUE
Here is what is in the container:
1 2 3 4
Setting entries to 0:
Container now contains:
0 0 0 0
Vector Iterator Types

• Iterators for vectors of ints are of type:
  
  std::vector<int>::iterator

• Iterators for lists of ints are of type:
  
  std::list<int>::iterator

• Vector is in std namespace, so need:
  
  using std::vector<int>::iterator;
Kinds of Iterators

- Different containers → different iterators
- Vector iterators
  - Most "general" form
  - All operations work with vector iterators
  - Vector container great for iterator examples
int main() {
    vector<char> container;
    container.push_back('A');
    container.push_back('B');
    container.push_back('C');
    container.push_back('D');

    for (int i = 0; i < 4; i++)
        cout << "container[" << i << "] == " << container[i] << endl;

    vector<char>::iterator p = container.begin();
    cout << "The third entry is " << container[2] << endl;
    cout << "The third entry is " << p[2] << endl;
    cout << "The third entry is " << *(p + 2) << endl;

    cout << "Back to container[0].\n";
    p = container.begin();
    cout << "which has value " << *p << endl;

    cout << "Two steps forward and one step back:\n";
    p++;
    cout << *p << endl;
Iterator Classifications

• Forward iterators:
  – ++ works on iterator

• Bidirectional iterators:
  – Both ++ and -- work on iterator

• Random-access iterators:
  – ++, --, and random access all work with iterator

• These are "kinds" of iterators, not types!
Constant and Mutable Iterators

• Dereferencing operator’s behavior dictates

• Constant iterator:
  – * produces read-only version of element
  – Can use *p to assign to variable or output, but cannot change element in container
    • E.g., *p = <anything>; is illegal

• Mutable iterator:
  – *p can be assigned value
  – Changes corresponding element in container
  – i.e.: *p returns an lvalue
Reverse Iterators

• To cycle elements in reverse order
  – Requires container with bidirectional iterators

• Might consider:
  iterator p;
  for (p=container.end(); p!=container.begin(); p--)
    cout << *p << " " ;
  – But recall: end() is just "sentinel", begin() not!
  – Might work on some systems, but not most
Reverse Iterators Correct

• To correctly cycle elements in reverse order:
  reverse_iterator p;
  for (rp=container.rbegin(); rp!=container.rend(); rp++)
    cout << *rp << " ";

• rbegin()
  – Returns iterator at last element

• rend()
  – Returns sentinel "end" marker
Compiler Problems

• Some compilers problematic with iterator declarations

• Consider our usage:
  using std::vector<char>::iterator;
  ...
  iterator p;

• Alternatively:
  std::vector<char>::iterator p;

• And others...
  – Try various forms if compiler problematic
Containers

• Container classes in STL
  – Different kinds of data structures
  – Like lists, queues, stacks

• Each is template class with parameter for particular data type to be stored
  – e.g., Lists of ints, doubles or myClass types

• Each has own iterators
  – One might have bidirectional, another might just have forward iterators

• But all operators and members have same meaning
Sequential Containers

• Arranges list data
  – 1\textsuperscript{st} element, next element, ... to last element

• Linked list is sequential container
  – Earlier linked lists were "singly linked lists"
    • One link per node

• STL has no "singly linked list"
  – Only "doubly linked list": template class \textit{list}
Display 19.4 Two Kinds of Lists

**slist**: A singly linked list
++ defined; -- not defined

**list**: A doubly linked list
Both ++ and -- defined

---

**slist** is not part of the STL and may not always be implemented. **list** is part of the STL.
Display 19.5
Using the list Template Class (1 of 2)

```cpp
// Program to demonstrate the STL template class list.
#include <iostream>
#include <list>
using std::cout;
using std::endl;
using std::list;

int main()
{
    list<int> listObject;
    for (int i = 1; i <= 3; i++)
        listObject.push_back(i);
    cout << "List contains: \n";
    list<int>::iterator iter;
    for (iter = listObject.begin(); iter != listObject.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}
```
cout << "Setting all entries to 0:\n";
for (iter = listObject.begin( ); iter != listObject.end( ); iter++)
    *iter = 0;

cout << "List now contains:\n";
for (iter = listObject.begin( ); iter != listObject.end( ); iter++)
    cout << *iter << " ";
cout << endl;

return 0;
}

SAMPLE DIALOGUE
List contains:
1 2 3
Setting all entries to 0:
List now contains:
0 0 0
Container Adapters stack and queue

• Container adapters are template classes
  – Implemented "on top of" other classes

• Example:
  *stack* template class by default implemented on top of *deque* template class
  – Buried in stack’s implementation is deque where all data resides

• Others:
  queue, priority_queue
Specifying Container Adapters

• Adapter template classes have "default" containers underneath
  – But can specify different underlying container
  – Examples:
    stack template class → any sequence container
    priority_queue → default is vector, could be others

• Implementing Example:
  stack<int, vector<int> >
  – Makes vector underlying container for stack
Generic Algorithms

• Basic template functions
• Recall algorithm definition:
  – Set of instructions for performing a task
  – Can be represented in any language
  – Typically thought of in "pseudocode"
  – Considered "abstraction" of code
    • Gives important details, but not find code details
• STL’s algorithms in template functions:
  – Certain details provided only
    • Therefore considered "generic algorithms"
Running Times

• How fast is program?
  – "Seconds"?
  – Consider: large input? .. small input?

• Produce "table"
  – Based on input size
  – Table called "function" in math
    • With arguments and return values!
  – Argument is input size:
    T(10), T(10,000), ...

• Function T is called "running time"
Consider Sorting Program

• Faster on smaller input set?
  – Perhaps
  – Might depend on "state" of set
    • "Mostly" sorted already?

• Consider worst-case running time
  – $T(N)$ is time taken by "hardest" list
    • List that takes longest to sort
Counting Operations

• T(N) given by formula, such as:
  \[ T(N) = 5N + 5 \]
  – "On inputs of size N program runs for 5N + 5 time units"

• Must be "computer-independent"
  – Doesn’t matter how "fast" computers are
  – Can’t count "time"
  – Instead count "operations"
Counting Operations Example

• int I = 0;
  bool found = false;
  while (( I < N) && !found)
    if (a[I] == target)
      found = true;
    else
      I++;

• 5 operations per loop iteration:
  <, &&, !, [ ], ==, ++

• After N iterations, final three: <, &&, !

• So: 6N+5 operations when target not found
Big-O Notation

• Recall: 6N+5 operations in "worst-case"
• Expressed in "Big-O" notation
  – Some constant "c" factor where c(6N+5) is actual running time
    • c different on different systems
  – We say code runs in time O(6N+5)
  – But typically only consider "highest term"
    • Term with highest exponent
  – O(N) here
Big-O Terminology

• Linear running time:
  – $O(N)$—directly proportional to input size $N$
• Quadratic running time:
  – $O(N^2)$
• Logarithmic running time:
  – $O(\log N)$
    • Typically "log base 2"
    • Very fast algorithms!
Display 19.16
Comparison of Running Times

Comparison of Running Times

\[ T(N) = 0.5N^2 \]
\[ T(N) = 2N \]
\[ T(N) = N + 2 \]
\[ T(N) = N \]

\( N \) (problem size)
Summary 1

• Iterator is "generalization" of a pointer
  – Used to move through elements of container

• Container classes with iterators have:
  – Member functions end() and begin() to assist cycling

• Main kinds of iterators:
  – Forward, bi-directional, random-access

• Given constant iterator p, *p is read-only version of element
Summary 2

- Given mutable iterator p → *p can be assigned value
- Bidirectional container has reverse iterators allowing reverse cycling
- Main STL containers: list, vector, deque
  - stack, queue: container adapter classes
- set, map, multiset, multimap containers store in sorted order
- STL implements generic algorithms
  - Provide maximum running time guarantees