The Standard Template Library (STL)

Learning Objectives

- Iterators
  - Constant and mutable iterators
  - Reverse iterators
- Containers
  - Sequential containers
  - Container adapters stack and queue
  - Associative Containers set and map
- Generic Algorithms
  - Sequence, set, and sorting algorithms

Introduction

- Custom data structures
  - You can 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: vectors and lists

Iterators

- An *iterator* is any object that, pointing to some element in a range of elements e.g. array, container
  - Lets you iterate through the elements of that range using a set of operators
    - Should at least provide the increment (++) and dereference (*) operators
  - "Abstraction" of iterators
    - Designed to hide details of implementation
    - Provide uniform interface across different container classes
- Recall: generalization of a pointer
  - Typically even implemented with pointer!
- 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

Display 19.1
Iterators Used with a Vector (1 of 2)

```
//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:\n";
    vector<int>::iterator p;
    for (p = container.begin( ); p != container.end( ); p++)
        cout << *p << " ";
    cout << endl;

    cout << "Setting entries to 0:\n";
    for (p = container.begin( ); p != container.end( ); p++)
         *p = 0;
```

Display 19.1
Iterators Used with a Vector (2 of 2)

```
    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 and list 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:

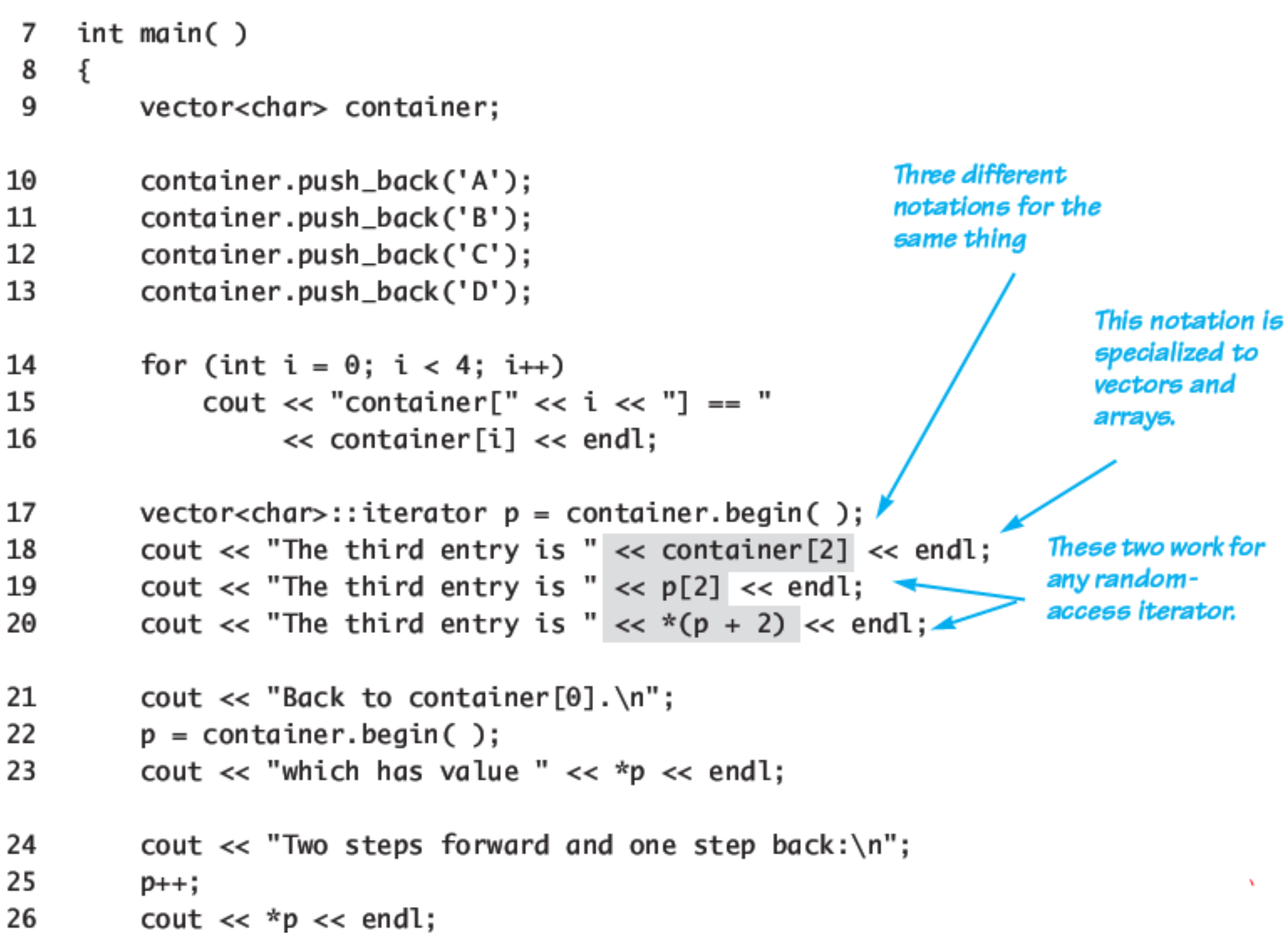
  ```
  using std::vector<int>::iterator;
  ```

Kinds of Iterators

- Different containers → different iterators
- Vector iterators
  - Most "general" form
  - All operations work with vector iterators
  - Vector container great for iterator examples

Random Access:

Display 19.2 Bidirectional and Random-Access Iterator Use



```
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;
```

Three different notations for the same thing

This notation is specialized to vectors and arrays.

These two work for any random-access iterator.

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 (const_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 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

Auto

- The C++11 **auto** keyword can make your code much more readable when it comes to templates and iterators.
- Instead of

```
    vector<int>::iterator p = v.begin();
```

- We can do the same thing much more compactly with auto

```
    auto p = v.begin();
```

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
  - 1st 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 *list*

Display 19.5
Using the list Template Class(1 of 2)

```
//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;
```

Display 19.5
Using the list Template Class(2 of 2)

```
    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
```

Associative Containers

- Associative container: simple database
- Store data
  - – Each data item has key
- Example:
  data: employee’s record as struct
  key: employee’s SSN
  - – Items retrieved based on key

set Template Class

- Simplest container possible
- Stores elements without repetition
- Designed to be efficient
  - Stores values in sorted order
- 1st insertion places element in set
- Each element is own key
- Capabilities:
  - Add elements
  - Delete elements
  - Ask if element is in set

Program Using the set Template Class (1 of 2)

```
//Program to demonstrate use of the set template class.
#include <iostream>
#include <set>
using std::cout;
using std::endl;
using std::set;

int main( )
{
    set<char> s;

    s.insert(’A’);
    s.insert(’D’);
    s.insert(’D’);
    s.insert(’C’);
    s.insert(’C’);
    s.insert(’B’);

    cout << "The set contains:\n";
    set<char>::const_iterator p;
    for (p = s.begin( ); p != s.end( ); p++)
    cout << *p << " ";
    cout << endl;
```

Program Using the set Template Class (2 of 2)

```
     cout << "Set contains 'C': ";
     if (s.find('C')==s.end( ))
        cout << " no " << endl;
     else
        cout << " yes " << endl;

        cout << "Removing C.\n";
        s.erase(’C’);
        for (p = s.begin( ); p != s.end( ); p++)
        cout << *p << " ";
        cout << endl;

     cout << "Set contains 'C': ";
     if (s.find('C')==s.end( ))
         cout << " no " << endl;
     else
         cout << " yes " << endl;

        return 0;
    }
```

SAMPLE DIALOGUE

```
The set contains:
A B C D
Set contains 'C':  yes
Removing C.
A B D
Set contains 'C':  no
```

Map Template Class

- A function given as set of ordered pairs
  - For each value first, at most one value second in map
- Example map declaration:
  map<string, int> numberMap;
- Can use [ ] notation to access the map
  - For both storage and retrieval
- Stores in sorted order, like set
  - Second value can have no ordering impact

Program Using the map Template Class (1 of 3)

```
//Program to demonstrate use of the map template class.
#include <iostream>
#include <map>
#include <string>
using std::cout;
using std::endl;
using std::map;
using std::string;

int main( )
{
    map<string, string> planets;

    planets["Mercury"] = "Hot planet";
    planets["Venus"] = "Atmosphere of sulfuric acid";
    planets["Earth"] = "Home";
    planets["Mars"] = "The Red Planet";
    planets["Jupiter"] = "Largest planet in our solar system";
    planets["Saturn"] = "Has rings";
    planets["Uranus"] = "Tilts on its side";
    planets["Neptune"] = "1500 mile per hour winds";
    planets["Pluto"] = "Dwarf planet";
```

Program Using the map Template Class (2 of 3)

```
    cout << "Entry for Mercury - " << planets["Mercury"]
            << endl << endl;

    if (planets.find("Mercury") != planets.end())
        cout << "Mercury is in the map." << endl;
    if (planets.find("Ceres") == planets.end())
        cout << "Ceres is not in the map." << endl << endl;

    cout << "Iterating through all planets: " << endl;
    map<string, string>::const_iterator iter;
    for (iter = planets.begin(); iter != planets.end(); iter++)
    {
        cout << iter->first << " - " << iter->second << endl;
    }
```

The iterator will output the map in order sorted by the key. In this case the output will be listed alphabetically by planet.

```
    return 0;
}
```

Program Using the map Template Class (3 of 3)

```
SAMPLE DIALOGUE

Entry for Mercury - Hot planet

Mercury is in the map.
Ceres is not in the map.

Iterating through all planets:
Earth - Home
Jupiter - Largest planet in our solar system
Mars - The Red Planet
Mercury - Hot planet
Neptune - 1500 mile per hour winds
Pluto - Dwarf planet
Saturn - Has rings
Uranus - Tilts on its side
Venus - Atmosphere of sulfuric acid
```

Use Initialization, Ranged For, and auto with Containers

- C++11’s ranged for, auto, and initialization features make it easier to work with Containers
- Consider:

```
map<int, string> personIDs = {
                                  {1,"Walt"},
                                  {2,"Kenrick"}
                                };
set<string> colors = {"red","green","blue"};
```

- We can easily iterate through each with:

```
for (auto p : personIDs)
       cout << p.first << " " << p.second << endl;
 for (auto p : colors)
       cout << p << " ";
```

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"

Nonmodifying Sequence Algorithms

- Template functions operating on containers
  - NO modification of container contents
- Generic find function
  - Typical example
  - Can be used with any STL sequence container class

Display 19.17

The Generic find Function (1 of 3)

```
//Program to demonstrate use of the generic find function.
#include <iostream>
#include <vector>
#include <algorithm>
using std::cin;
using std::cout;
using std::endl;
using std::vector;
using std::find;

int main( )
{
    vector<char> line;

    cout << "Enter a line of text:\n";
    char next;
    cin.get(next);
    while (next != ’\n’)
    {
        line.push_back(next);
        cin.get(next);
    }
```

Display 19.17

The Generic find Function (2 of 3)

```
        vector<char>::const_iterator where;
        where = find(line.begin( ), line.end( ), ’e’);
        //where is located at the first occurrence of ’e’ in v.

        vector<char>::const_iterator p;
        cout << "You entered the following before you entered your
              first e:\n";
        for (p = line.begin( ); p != where; p++)
            cout << *p;
        cout << endl;

        cout << "You entered the following after that:\n";
        for (p = where; p != line.end( ); p++)
            cout << *p;
        cout << endl;

        cout << "End of demonstration.\n";
        return 0;
}
```

If find does not find what it is looking for, it returns its second argument.

Display 19.17
The Generic find Function (3 of 3)

SAMPLE DIALOGUE 1

```
Enter a line of text
A line of text.
You entered the following before you entered your first e:
A lin
You entered the following after that:
e of text.
End of demonstration.
```

SAMPLE DIALOGUE 2

```
Enter a line of text
I will not!
You entered the following before you entered your first e:
I will not!
You entered the following after that:

End of demonstration.
```

Modifying Sequence Algorithms

- STL functions that change container contents
- Recall: adding/removing elements from containers can affect other iterators!
  - list, slist guarantee no iterator changes
  - vector, deque make NO such guarantee
- Always watch which iterators are assured to be changed/unchanged

Set Algorithms

- STL generic set operation functions
- All assume containers stored in sorted order
- Containers set, map, multiset, multimap
  - DO store in sorted order, so all set functions apply
- Others, like vector, are not sorted
  - Should not use set functions

Sorting Algorithms

```
#include <algorithm>
#include <vector>
#include <iostream>

int main(){
    std::vector<int> s = {5, 7, 4, 2, 8, 6, 1, 9, 0,78,-6,77, 3};
    // sort using the default operator<
    std::sort(s.begin(), s.end());
```

- STL contains two template functions:
  1. sort range of elements
  2. merge two sorted ranges of elements
- Guaranteed running time O(N log N)
  – No sort can be faster
  – Function guarantees fastest possible sort