



GLOBAL
EDITION



Chapter 19

Standard Template Library

Absolute C++

SIXTH EDITION
Walter Savitch

ALWAYS LEARNING

PEARSON

Copyright © 2017 Pearson Education, Ltd.
All rights reserved.

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

Display 19.1

Iterators Used with a Vector (1 of 2)

```
1 //Program to demonstrate STL iterators.
2 #include <iostream>
3 #include <vector>
4 using std::cout;
5 using std::endl;
6 using std::vector;

7 int main( )
8 {
9     vector<int> container;

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

12    cout << "Here is what is in the container:\n";
13    vector<int>::iterator p;
14    for (p = container.begin( ); p != container.end( ); p++)
15        cout << *p << " ";
16    cout << endl;

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

Display 19.1

Iterators Used with a Vector (2 of 2)

```
20         cout << "Container now contains:\n";
21         for (p = container.begin( ); p !=
                container.end( ); p++)
22             cout << *p << " ";
23         cout << endl;

24         return 0;
25     }
```

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

Random Access:

Display 19.2 Bidirectional and Random-Access Iterator Use

```
7  int main( )
8  {
9      vector<char> container;

10     container.push_back('A');
11     container.push_back('B');
12     container.push_back('C');
13     container.push_back('D');

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

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

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

24     cout << "Two steps forward and one step back:\n";
25     p++;
26     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:
 - * 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

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

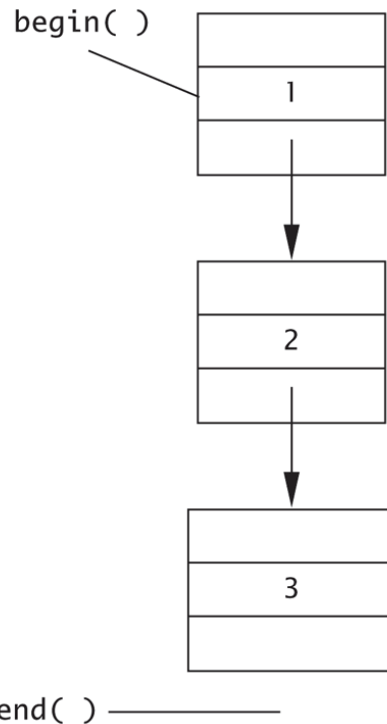
Sequential Containers

- 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*

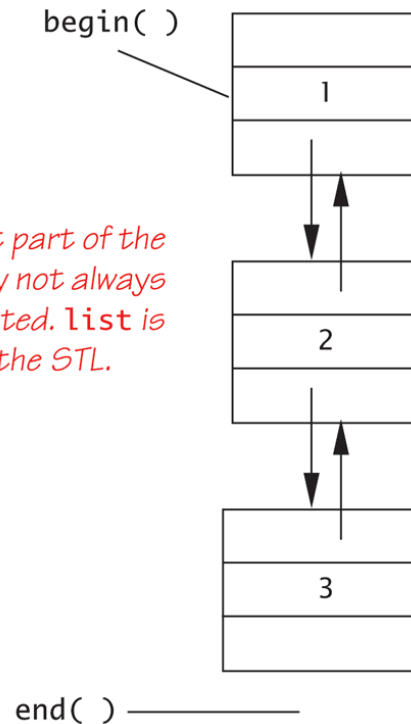
Display 19.4 Two Kinds of Lists

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)

```
1      //Program to demonstrate the STL template class list.
2      #include <iostream>
3      #include <list>
4      using std::cout;
5      using std::endl;
6      using std::list;

7      int main( )
8      {
9          list<int> listObject;

10         for (int i = 1; i <= 3; i++)
11             listObject.push_back(i);

12         cout << "List contains:\n";
13         list<int>::iterator iter;
14         for (iter = listObject.begin( ); iter != listObject.end( );
15             iter++)
16             cout << *iter << " ";
17         cout << endl;
```

Display 19.5

Using the list Template Class(2 of 2)

```
17         cout << "Setting all entries to 0:\n";
18         for (iter = listObject.begin( ); iter != listObject.end( );
               iter++)
19             *iter = 0;

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

24         return 0;
25     }
```

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

 Note space between > >

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
- 1st insertion places element in set
- Each element is own key
- Capabilities:
 - Add elements
 - Delete elements
 - Ask if element is in set

More set Template Class

- Designed to be efficient
 - Stores values in sorted order
 - Can specify order:
`set<T, Ordering> s;`
 - Ordering is well-behaved ordering relation that returns bool
 - None specified: use < relational operator

Program Using the set Template Class (1 of 2)

```
1      //Program to demonstrate use of the set template class.
2      #include <iostream>
3      #include <set>
4      using std::cout;
5      using std::endl;
6      using std::set;

7      int main( )
8      {
9          set<char> s;

10         s.insert('A');
11         s.insert('D');
12         s.insert('D');
13         s.insert('C');
14         s.insert('C');
15         s.insert('B');

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

Program Using the set Template Class (2 of 2)

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

27      cout << "Removing C.\n";
28      s.erase('C');
29      for (p = s.begin( ); p != s.end( ); p++)
30          cout << *p << " ";
31      cout << endl;
```

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

37      return 0;
38  }
```

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)

```
1      //Program to demonstrate use of the map template class.
2      #include <iostream>
3      #include <map>
4      #include <string>
5      using std::cout;
6      using std::endl;
7      using std::map;
8      using std::string;

9      int main( )
10     {
11         map<string, string> planets;

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

Program Using the map Template Class (2 of 3)

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

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

27         cout << "Iterating through all planets: " << endl;
28         map<string, string>::const_iterator iter;
29         for (iter = planets.begin(); iter != planets.end(); iter++)
30             {
31                 cout << iter->first << " - " << iter->second << endl;
32             }
```

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

```
33         return 0;
34     }
```


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 << " ";
```

Efficiency

- STL designed with efficiency as important consideration
 - Strives to be optimally efficient
- Example: set, map elements stored in sorted order for fast searches
- Template class member functions:
 - Guaranteed maximum running time
 - Called "Big-O" notation, an "efficiency"-rating

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 fine 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"

Table for Running Time Function:

Display 19.15 Some Values of a Running Time Function

Some Values of a Running Time Function

INPUT SIZE	RUNNING TIME
10 numbers	2 seconds
100 numbers	2.1 seconds
1,000 numbers	10 seconds
10,000 numbers	2.5 minutes

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 l = 0;
bool found = false;
while ((l < N) && !found)
 if (a[l] == target)
 found = true;
 else
 l++;
```
- 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

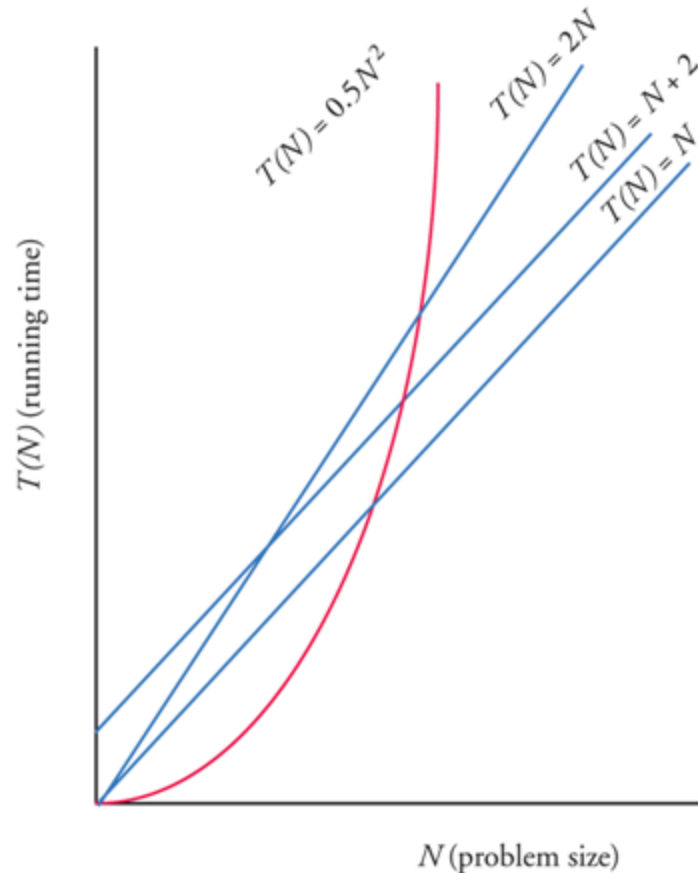
- Linear running time:
  - $O(N)$ —directly proportional to input size  $N$
- Quadratic running time:
  - $O(N^2)$
- Logarithmic running time:
  - $O(\log N)$ 
    - Typically "log base 2"
    - Very fast algorithms!

# Display 19.16

## Comparison of Running Times

Comparison of Running Times

---



# Container Access Running Times

- $O(1)$  - constant operation always:
  - Vector inserts to front or back
  - deque inserts
  - list inserts
- $O(N)$ 
  - Insert or delete of arbitrary element in vector or deque (N is number of elements)
- $O(\log N)$ 
  - set or map finding

# 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)

```
1 //Program to demonstrate use of the generic find function.
2 #include <iostream>
3 #include <vector>
4 #include <algorithm>
5 using std::cin;
6 using std::cout;
7 using std::endl;
8 using std::vector;
9 using std::find;

10 int main()
11 {
12 vector<char> line;

13 cout << "Enter a line of text:\n";
14 char next;
15 cin.get(next);
16 while (next != '\n')
17 {
18 line.push_back(next);
19 cin.get(next);
20 }
```

# Display 19.17

## The Generic find Function (2 of 3)

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

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

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

33 cout << "End of demonstration.\n";
34 return 0;
35 }
```

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

- 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

# 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  $p \rightarrow *p$  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