Standard Template Library

Modified from Chapter 18

Standard Template Library

- **STL has containers, algorithms and Iterators**
  - Containers hold objects, all of a specified type
  - Generic algorithms act on objects in containers
  - Iterators provide access to objects in the containers yet hide the internal structure of the container

Iterators

Modified from Section 18.1

Iterator Basics

- An **iterator** is a generalization of pointer
  - Not a pointer but usually implemented using pointers
  - The pointer operations may be overloaded for behaviour appropriate for the container internals
  - Treating iterators as pointers typically is OK.
  - Each container defines an appropriate iterator type.
  - Operations are consistent across all iterator types.

Basic Iterator Operations

- Basic operations shared by all iterator types
  - ++ (pre- and postfix) to advance to the next data item
  - == and != operators to test whether two iterators point to the same data item
  - * dereferencing operator provides data item access
  - **c.begin( )** returns an iterator pointing to the first element of container **c**
  - **c.end( )** returns an iterator pointing **past** the last element of container **c. Analogous to the null pointer. Unlike the null pointer, you can apply -- to the iterator returned by c.end( ) to get an iterator pointing to last element in the container.**

More Iterator Operations

- **--** (pre- and postfix) moves to previous data item Available to some kinds of iterators.
- ***p** access may be read-only or read-write depending on the container and the definition of the iterator p.
- STL containers define iterator types appropriate to the container internals.
- Some containers provide read-only iterators

Display 18.1

Display 18.1

Back Next

DISPLAY 18.1 Iterators Used with a Vector

```
//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;

    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
```

Kinds of Iterators

- Forward iterators provide the basic operations
- Bidirectional iterators provide the basic operations and the -- operators (pre- and postfix) to move to the previous data item.
- Random access iterators provide
  - The basic operations and --
  - Indexing
    - **p[2]** returns the third element in the container
  - Iterator arithmetic
    - **p + 2** returns an iterator to the third element in the container
    - **c.end()-1** returns an iterator to the last element in the container

Constant and Mutable Iterators

- Categories of iterator divide into constant and mutable iterator.
  - Constant Iterator **cp** does not allow assigning element at **cp**

    using std::vector<int>::const_iterator;
    const_iterator cp = v.begin( );
    ***cp = something; // illegal**
  - Mutable iterator **p** does allow changing the element at **p**.

    using std::vector<int>::iterator;
    iterator p = v.begin( );
    ***p = something; // OK**

Using auto

- The C++11 auto keyword can simplify variable declarations for iterators, e.g. instead of:

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

- We simply use:

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

Reverse Iterators

- A reverse iterator enables cycling through a container from the end to the beginning. Reverse iterators reverse the more usual behaviour of ++ and --
- **rp++** moves the reverse iterator **rp** towards the beginning of the container.
- **rp--** moves the reverse iterator **rp** towards the end of the container.

```
reverse_iterator rp;
for(rp = c.rbegin( ); rp != c.rend( ); rp++)
    process_item_at (rp);
```

Object **c** is a container with bidirectional iterators

Containers

Modified from Section 18.2

Containers

- The STL provides three kinds of containers:
  - Sequential Containers are containers where the ultimate position of the element depends on where it was inserted, not on its value.
    - e.g. list, vector, deque
  - Container Adapters use the sequential containers for storage, but modify the user interface to stack, queue or other structure.
    - e.g. stack, queue
  - Associative Containers maintain the data in sorted order to implement the container's purpose. The position depends on the value of the element.
    - e.g. set, map

Sequential Containers

- The STL sequential containers are the **list**, **vector** and **deque.**
- Sequential means the container has a first, element, a second element and so on.
- An STL **list** is a doubly linked list.
- An STL **vector** is essentially an array whose allocated space can grow while the program runs.
- An STL **deque** ("d-que" or "deck") is a "double ended queue". Data can be added or removed at either end and the size can change while the program runs.

Common Container Members

- The STL sequential containers each have different characteristics, but they *all* support these members:
  - ***container*( ); // creates empty container e.g. vector()**
  - **~*container*( ); // destroys container, erases all members**
  - **c.empty( ) // true if there are no entries in c**
  - **c.size( ) const; // number of entries in container c**
  - **c = v; //replace contents of c with contents of v**

More Common Container Members

- **c.swap(other_container); // swaps contents of**
  // c and *other_container*.
- **c.push_back(item); // appends item to container c**
- **c.begin( ); // returns an iterator to the first**
  // element in container c
- **c.end( ); // returns an iterator to a position**
  // beyond the end of the container c.
- **c.rbegin( ); // returns an iterator to the last element**
  // in the container. Serves to as start of
  // reverse traversal.

More Common Container Members

- **c.rend( ); // returns an iterator to a position**
  // beyond the front of the container.
- **c.front( ); // returns the first element in the**
  // container (same as *c.begin( );)
- **c.back( ); //returns the last element in the container**
  // same as *(--c.end( ));
- **c.insert(iter, elem); //insert copy of element *elem***
  //before iter
- **c.erase(iter); //removes the element that *iter* points to,**
  // returns an iterator to element
  // following erasure. returns c.end( ) if
  // last element is removed.

More Common Container Members

- **c.clear( ); // makes container c empty**
- **c1 == c2 // returns true if the sizes equal and**
  // corresponding elements in c1 and c2 are
  //equal
- **c1 != c2 // returns !(c1==c2)**
- **c.push_front(elem) // insert element *elem* at the**
  // front of container c.
  // NOT implemented for *vector* due to large
  // run-time that results

PITFALL:
Iterators and Removing and Inserting Elements

- Removing elements will invalidate some iterators.
- **erase** member function returns an iterator pointing to the next element past the erased element.
- With **list** we are guaranteed that only iterators pointing to the erased element are invalidated.
- With **vector** and **deque,** treat all operations that erase or insert as invalidating previously defined iterators.

PITFALL:
Iterators and Removing Elements

```
vector<int> v(10);
for(int i = 0; i < 10; i++)
        v[i] = i+1;

auto it = v.begin()+1;
auto last = v.end() - 1;

v.erase(it);

// what is the output?
for(auto i = last; i != v.end(); i++)
        cout << *i << " ";
cout << endl;
```

PITFALL:
Iterators and Inserting Elements

```
vector<int> v(5);
for(int i = 0; i < 5; i++)
        v[i] = i+1;

auto first = v.begin();

for(int i = 0; i < 1000; i++)
{
        auto it = v.begin()+1;
        v.insert(it, 100);
}

// what is the output?
for(auto i = first; i != v.end(); i++)
        cout << *i << " ";
cout << endl;
```

Operation Support

Operation	Function	vector	List	deque
Insert at front	push_front(e)	-	X	X
Insert at back	push_back(e)	X	X	X
Delete at front	pop_front( )	-	X	X
Delete at back	pop_back( )	X	X	X
Insert in middle	insert(iter, e)	(X)	X	(X)
Delete in middle	erase(iter)	(X)	X	(X)
Sort	sort( )	X	-	X

(X) Indicates this operation is significantly slower.

Display 18.6

Display 18.6

Back Next

DISPLAY 18.6 STL Basic Sequential Containers

Template Class Name	Iterator Type Names	Kind of Iterators	Library Header File
slist Warning: slist is not part of the STL.	slist<T>::iterator slist<T>::const_iterator	mutable forward constant forward	<slist> Depends on implementation and may not be available.
list	list<T>::iterator list<T>::const_iterator list<T>::reverse_iterator list<T>::const_reverse_iterator	mutable bidirectional constant bidirectional mutable bidirectional constant bidirectional	<list>
vector	vector<T>::iterator vector<T>::const_iterator vector<T>::reverse_iterator vector<T>::const_reverse_iterator	mutable random access constant random access mutable random access constant random access	<vector>
deque	deque<T>::iterator deque<T>::const_iterator deque<T>::reverse_iterator deque<T>::const_reverse_iterator	mutable random access constant random access mutable random access constant random access	<deque>

The Container Adapters **stack** and **queue**

- Container Adapters use sequence containers for storage but supply a different user interface.
- A **stack** uses a Last-In-First-Out discipline.
- A **queue** uses a First-In-First-Out discipline.
- The **deque** is the default container for both stack and queue.

Container Adapter **stack**

- Declarations:
  - **stack<T> s; // uses deque as underlying store**
  - **stack<T, underlying_container> t ; //uses the specified // container as underlying container for stack**
  - **stack<T> s (sequence_container); // initializes stack to**

    // elements in sequence_container.
- Header:

  #include <stack>
- Defined types:

  value_type, size_type
- No iterators are defined.

stack Member Functions

Sample Member Functions	
Member function	Returns
s.size( )	number of elements in stack
s.empty( )	true if no elements in stack else false
s.top( )	reference to top stack member
s.push(elem)	void Inserts copy of *elem* on stack top
s.pop( )	void function. Removes top of stack.
s1 == s2	true if sizes same and corresponding pairs of elements are equal, else false

Container Adapter **queue**

- Declarations:
  - **queue<T> q; // uses deque as underlying store**
  - **queue<T, underlying_container> q ; //uses the specified //container as underlying container for queue**
  - **queue<T> s (sequence_container); // initializes queue to**
  - **// elements in sequence_container.**
- Header:
- **#include <queue>**
- Defined types:
- **value_type, size_type**
- No iterators are defined.

queue Member Functions

Sample Member Functions	
Member function	Returns
q.size( )	number of elements in queue
q.empty( )	true if no elements in queue else false
q.front( )	reference to front queue member
q.push(elem)	void adds a copy of *elem* at queue rear
q.pop( )	void function. Removes front of queue.
q1 == q2	true if sizes same and corresonding pairs of elements are equal, else false

Associative Containers set and map

- **Associative containers** keep elements sorted on some property of the element called the **key**.
- Only the first insertion of a value into a **set** has effect. (like a mathematical set, does not support duplicate values)
- The default order is the **<** relational operator for both **set** and **map**.

The **set** Associative Container

- Declarations:
  - **set<T> s; // uses deque as underlying store**
  - **set<T, Ordering> s ; //uses the specified**
    // order relation to sort elements in the set
    // uses < if no order is specified.
- Header:

  #include <set>
- Defined types:

  value_type, size_type
- Iterators: **iterator**, **const_iterator**, **reverse_iterator**, **const_reverse_iterator**

set Member Functions

function	Returns
s.size( )	number of elements in set
s.empty( )	true if no elements in set else false
s.insert(el)	Insert *elem* in set. No effect if *el* is a member
s.erase(itr)	Erase element to which *itr* refers
s.erase(el)	Erase element *el* from set. No effect if *el* is not a member
s.find(el)	Mutable iterator to location of *el* in set if present, else returns s.end( )
s1 == s2	true if sizes same and corresponding pairs of elements are equal, else false

Display 18.12

Display 18.12

DISPLAY 18.12 Program Using the set Template Class

```
//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;

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

    return 0;
}
```

No matter how many times you add an element to a set, the set contains only one copy of that element.

Sample Dialogue

```
The set contains:
A B C D
Removing C.
A B D
```

The **map** associative container

- A **map** is a set of ordered pairs **<first, second>**
- For each **first** in an ordered pair **<first, second>** there is at most one value, **second**, that appears in an ordered pair in the **map**.
  - **First** and **second** can be different data types, so for example you could map an integer to a string
- The STL provides a template class **pair<T1,T2>** defined in the **utility** header file.
- You may wish to read about the **multiset** and **multimap.** See Josuttis, The C++ Standard Library Addison Wesley.

map Member Functions

Function	Returns
m.size( )	number of pairs in the map
m.empty( )	true if no pairs are in the map else false
m.insert(el) *el* is a **pair<key, T>**	Inserts *el* into map. Returns <iterator, bool>. If successful, bool is true, iterator points to inserted pair. Otherwise bool is false
m.erase(key)	Erase element with key value *key* from map.
m.find(el)	Mutable iterator to location of *el* in map if present, else returns m.end( )
m1 == m2	true if maps contain the same pairs, else false
m[target]	Returns a reference to the map object associated to a key of target.

Maps as associative arrays

- An alternative interpretation is that a map is an associative array. For example,
  `numbermap["c++"] = 5`
  associates the integer 5 with the string "c++"
- The easiest way to add and retrieve data from a map is to use the [] operator
  - However, if you attempt to access map[key] and key is not already in the map, then a new entry with the default value will be added!

Display 18.14 (1-2)

Display 18.14 (1/2)

Back Next

DISPLAY 18.14 Program Using the `map` Template Class *(part 1 of 2)*

```
//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";

    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;
```

Display 18.14 (2/2)

Back Next

DISPLAY 18.14 Program Using the `map` Template Class *(part 2 of 2)*

```
    for (iter = planets.begin(); iter != planets.end(); iter++)
    {
      cout << iter->first  << " - " << iter->second << endl;
    }
    return 0;
}
```

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

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
```

Generic Algorithms

Modified from Section 18.3

Generic Algorithms

- “Generic Algorithm” are template functions that use iterators as template parameters.
- This chapter will use *Generic Algorithm*, *Generic function*, and *STL function template* to mean the same thing.
- Function interface specifies task, minimum strength of iterator arguments, and provides run-time specification.

Generic Algorithm Classification

- Algorithms are classified into
  - Nonmodifying Sequence Algorithms
  - Modifying Sequence Algorithms
  - Sorting and Related Algorithms
  - Numeric Algorithms
- We will deal with nonmodifying, modifying and sorting algorithms.
- Functions defined in *algorithm* library

  #include <algorithm>

Nonmodifying Sequence Algorithms

- Nonmodifying algorithms that do not modify the container they operate upon.
- The declaration of the generic function find algorithm:

  ```
  template<class ForwardIterator, class T>
  ForwardIterator find (ForwardIterator first, ForwardIterator last,
                                                  const T& value);
  ```

- The declaration tells us **find** works with any container that provides an iterator at least as strong as an input iterator.
- Type T objects must be equality comparable.

Iter find(Iter first, Iter last, const T& value);

- The generic algorithm **find( )** locates an element within a sequence. It takes three arguments.
- The first two specify a range: **[start, end)**, the third specifies a target **value** for the search.
- If requested value is found, **find( )** returns an iterator that points to the first element that is identical to the sought-after **value**.
- If the requested value is not found, **find( )** returns an iterator pointing one element past the final element in the sequence (that is, it returns the same value as **end( )** does).

More nonmodifying Algorithms

- **count** Counts occurrences of a value in a sequence
- **equal** Asks “are elements in two ranges equal?”
- **search** Looks for the first occurrence of a match sequence within another sequence
- **binary_search** Searches for a value in a container sorted using less. If the container was sorted using another predicate, this predicate must be supplied to binary_search. This is an efficient search for sorted sequences with random access iterators. Returns true or false.

Container Modifying Algorithms

- Container modifying algorithms change the content of the elements or their order.
  - **copy** Copies from a source range to a destination range. This can be used to shift elements in a container to the left
  - **remove** Removes all elements from a range equal to the given value
  - **random_shuffle** shuffles the elements of a sequence. Inputs must be Random Access Iterators
- For all functions one argument is an iterator pointing to the first of the sequence. Another argument points *one position past* the last element of the sequence.

Sorting Algorithm

- **sort** Sorts elements in a range in nondescending order, or in an order determined by a user-specified binary predicate.
  - template <class RandomAccessIterator>
    void sort (RandomAccessIterator first,
    RandomAccessIterator last);