C++

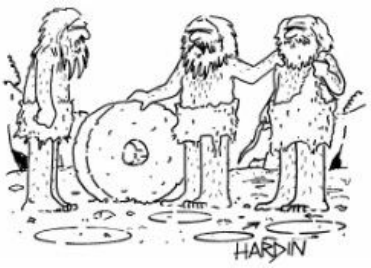
Standard Template Library

Outline

- Standard Template Library
- Containers & Iterators
- STL vector
- STL list
- STL stack
- STL queue

Software Engineering Observation

- Avoid reinventing the wheel; program with the reusable components of the C++ Standard Library.



Standard ***T***emplate ***L***ibrary

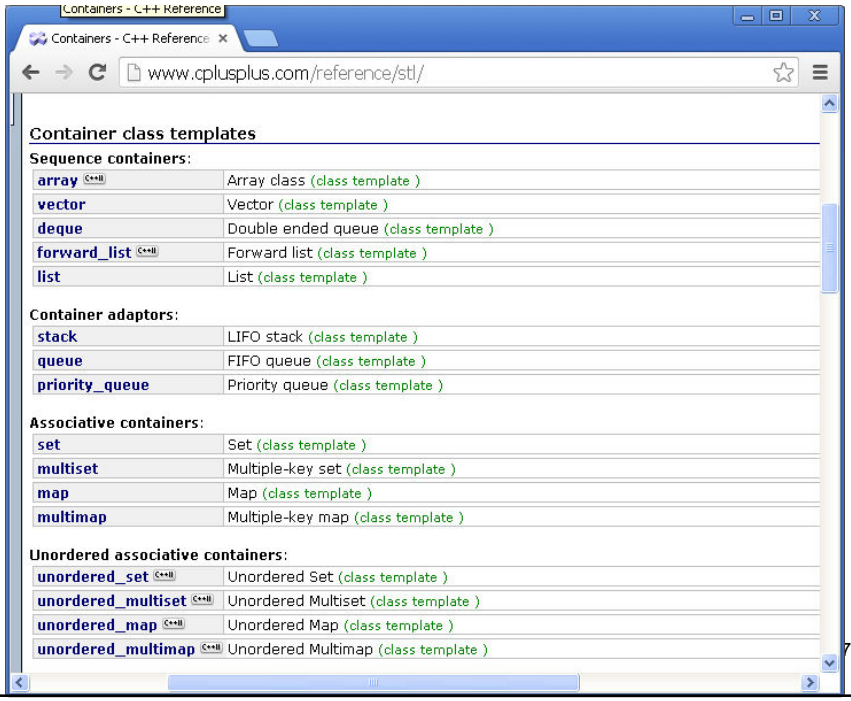
- The Standard Library is a fundamental part of the C++ Standard.
  - a comprehensive set of efficiently implemented tools and facilities.
- **S**tandard **T**emplate **L**ibrary (STL), which is the most important section of the Standard Library.

Standard Template Library (STL)

- Defines powerful, template-based, reusable components and algorithms to process them
  - Implement many common data structures
- Conceived and designed for performance and flexibility
- Three key components
  - Containers
  - Iterators
  - Algorithms

STL Containers

- A container is an object that represents a group of elements of a certain type, stored in a way that depends on the type of container (i.e., array, linked list, etc.).
- STL container: a generic type of container
  - the operations and element manipulations are identical regardless the type of underlying container that you are using.



STL Containers

- Type requirements for STL container elements
  - Elements must be copied to be inserted in a container
    - Element's type must provide copy constructor and assignment operator
    - Compiler will provide default memberwise copy and default memberwise assignment, which may or may not be appropriate
  - Elements might need to be compared
    - Element's type should provide equality operator and less-than operator

Iterators

- An iterator is a pointer-like object that is able to "point" to a specific element in the container.
- Iterators are often used to iterate over a range of objects:
  - if an iterator points to one element in a range, then it is possible to increment it so that it points to the next element.
- The iterator is container-specific.
  - The iterator's operations depend on what type of container we are using.

Iterators

- `iterator` versus `const_iterator`
  - `const_iterator`s cannot modify container elements

Predefined `typedef`s for iterator types	Direction of ++	Capability
`iterator`	forward	read/write
`const_iterator`	forward	read
`reverse_iterator`	backward	read/write
`const_reverse_iterator`	backward	read

Fig. 23.9 | Iterator `typedef`s.

STL sequence containers

- Three sequence containers
  - **`vector`** – a more robust type of array
  - **`list`** – implements a linked-list data structure
  - `deque` – based on arrays
- Common operations of sequence containers
  - **`front`** returns reference to first element
  - **`back`** returns reference to last element
  - **`push_back`** inserts new element to the end
  - **`pop_back`** removes last element

`vector`

- `vector` is a kind of sequence container. As such, its elements are ordered following a strict linear sequence.
- Vector containers are implemented as dynamic arrays;
  - The elements are stored in contiguous storage locations
  - Storage in vectors could be expanded and contracted as needed

vector

- Available to anyone building applications with C++

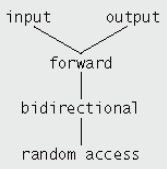
```
#include <vector>
  using std::vector;
```

- Can be defined to store any data type
  - Specified between angle brackets in **vector<*type*>**
  - All elements in a **vector** are set to 0 by default

```
vector<int> integers;
```


Iterator of Vector

- A vector supports random-access iterators.



Category	Description
input	Used to read an element from a container. An input iterator can move only in the forward direction (i.e., from the beginning of the container to the end) one element at a time. Input iterators support only one-pass algorithms—the same input iterator cannot be used to pass through a sequence twice.
output	Used to write an element to a container. An output iterator can move only in the forward direction one element at a time. Output iterators support only one-pass algorithms—the same output iterator cannot be used to pass through a sequence twice.
forward	Combines the capabilities of input and output iterators and retains their position in the container (as state information).
bidirectional	Combines the capabilities of a forward iterator with the ability to move in the backward direction (i.e., from the end of the container toward the beginning). Bidirectional iterators support multipass algorithms.
random access	Combines the capabilities of a bidirectional iterator with the ability to directly access any element of the container, i.e., to jump forward or backward by an arbitrary number of elements.

`Iterator` of `Vector`

- Member function **`begin():`** Returns an iterator referring to the first element.

```
iterator begin ();
const_iterator begin () const;
```

```
vector<int> integers;
const vector<int> constantIntegers;

vector<int>::iterator itStart;
vector<int>::const_iterator constItStart;

itStart = integers.begin();
constItStart = integers.begin();
constItStart = constantIntegers.begin();
```

Iterator of Vector

- Member function **end():** Returns an iterator referring to the *past-the-end* element.

```
iterator end ();
const_iterator end () const;
```

```
vector<int> integers;
const vector<int> constantIntegers;

vector<int>::iterator itEnd;
vector<int>::const_iterator constItEnd;

itEnd = integers.end();
constItEnd = integers.end();
constItEnd = constantIntegers.end();
```


Reverse Iterator of Vector

- Member function **rbegin():** Returns a reverse iterator referring to the last element.

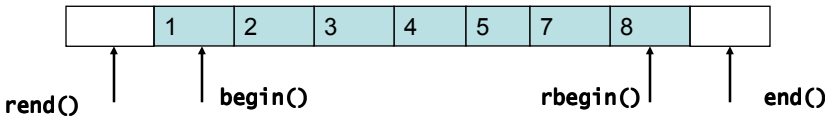
```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

- Member function **rend():** Returns a reverse iterator referring to the vector's *reverse end* - the element right before the first element

```
reverse_iterator rend();
const_reverse_iterator rend() const;
```

Iterator vs. Reverse Iterator

- **`rbegin`** refers to the element right before the one that would be referred to by member **`end()`**
- **`rend`** refers to the element right before the one that would be referred to by member **`begin()`**



Iterator operation	Description
`++p`	Preincrement an iterator.
`p++`	Postincrement an iterator.
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
`p == p1`	Compare iterators for equality.
`p != p1`	Compare iterators for inequality.
`--p`	Predecrement an iterator.
`p--`	Postdecrement an iterator.

Iterator operation	Description
`p += i`	Increment the iterator `p` by `i` positions.
`p -= i`	Decrement the iterator `p` by `i` positions.
`p + i`	Expression value is an iterator positioned at `p` incremented by `i` positions.
`p - i`	Expression value is an iterator positioned at `p` decremented by `i` positions.
`p[ i ]`	Return a reference to the element offset from `p` by `i` positions
`p < p1`	Return `true` if iterator `p` is less than iterator `p1` (i.e., iterator `p` is before iterator `p1` in the container); otherwise, return `false`.
`p <= p1`	Return `true` if iterator `p` is less than or equal to iterator `p1` (i.e., iterator `p` is before iterator `p1` or at the same location as iterator `p1` in the container); otherwise, return `false`.
`p > p1`	Return `true` if iterator `p` is greater than iterator `p1` (i.e., iterator `p` is after iterator `p1` in the container); otherwise, return `false`.
`p >= p1`	Return `true` if iterator `p` is greater than or equal to iterator `p1` (i.e., iterator `p` is after iterator `p1` or at the same location as iterator `p1` in the container); otherwise, return `false`.

Error-Prevention Tip

- The * (dereferencing) operator of any `const` iterator returns a `const` reference to the container element, disallowing the use of non-`const` member functions.

Common Programming Error

- Attempting to dereference an iterator positioned outside its container is a runtime logic error. In particular, the iterator returned by end cannot be dereferenced or incremented.

Constructors

```
• vector<int> first;
                          // empty vector of ints
• vector<int> second (4,100);
                          //four ints with value 100
• vector<int> third (second.begin(),second.end());
                          //iterating through second
• vector<int> fourth (third);
                              //a copy of third
• int myints[] = {16,2,77,29};
vector<int> fifth (myints,
             myints + sizeof(myints)/sizeof(int) );
```

Vector

- `int size() const;`
  - obtains size of array
- `int capacity() const;`
  - determines the amount of storage space they have allocated, and which can be either equal or greater than the actual size.
- `int max_size( ) const;`
  - determines the maximum number of elements that the vector container can hold.
- `bool empty() const;`
  - returns true is size of the vector is zero.

Add & Delete Element

- `void push_back(const T& element)`
  - Add element at the end

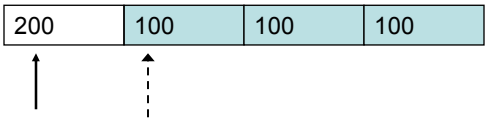
  `integers.push_back(1);`
- `void pop_back()`
  - Delete the last element

  `integers.pop_back();`

Insert Element

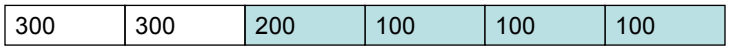
- Member function **`insert`**
  - The vector is extended by inserting new elements **before** the element at *position*.

```
vector<int> myvector (3,100);
vector<int>::iterator it = myvector.begin();
it = myvector.insert ( it , 200 );
//returns an iterator that points to the newly
  inserted element
```

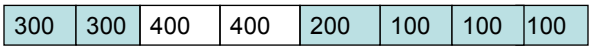


Insert Element

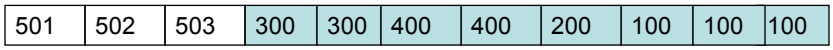
```
myvector.insert (it,2,300);
```




```
it = myvector.begin();
vector<int> anothervector (2,400);
myvector.insert (it+2, anothervector.begin( ),
                       anothervector.end());
```




```
int array [] = { 501,502,503 };
myvector.insert(myvector.begin(), array,array+3);
```



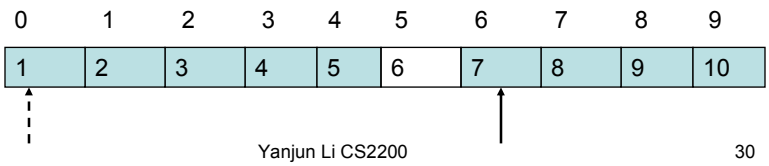
Performance Tip

- Insertion at the back of a `vector` is efficient. The **`vector`** simply grows, if necessary, to accommodate the new item. It is expensive to insert (or delete) an element in the middle of a **`vector`**—the entire portion of the `vector` after the insertion (or deletion) point must be moved, because `vector` elements occupy contiguous cells in memory just as C or C++ "raw" arrays do.

Erase Element

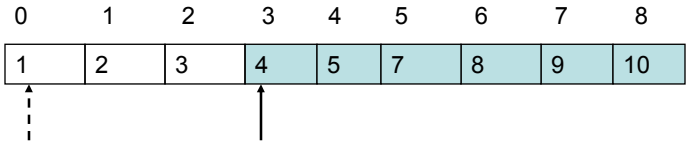
- Member function **`erase`**
  - Removes from the vector container either a single element - *position* or a range of elements - **[**first,last**)** .

```
for (i=1; i<=10; i++) myvector.push_back(i);
//erase the 6th element
myvector.erase (myvector.begin()+5);
```



Erase Element

```
//erase the first 3 elements:
myvector.erase (myvector.begin(),
                myvector.begin()+3);
```



Clear Content

- Member function **`clear`**
  - All the elements of the vector are dropped: their destructors are called, and then they are removed from the **`vector`** container, leaving the container with a **`size`** of 0.

```
myvector.clear();
```

Access `vector`'s Element

- `vector` member function `operator[ ]`
  - Not perform bounds checking
- `vector` member function `at`
  - Provides access to individual elements
  - Performs bounds checking
    - Throws an exception when specified index is invalid

Access `vector`'s Element

- Vector member function **`front()`**
  - Returns a reference to the first element in the vector container
- Vector member function **`back()`**
  - Returns a reference to the last element in the vector container
- The `vector` must not be empty; otherwise, results of the `front` and `back` functions are undefined.

Access `vector`'s Element

```
vector<int> myvector;
myvector.push_back(10);

while (myvector.back() != 0)
{
  myvector.push_back (myvector.back() -1 );
}
cout << "myvector contains:";
for (unsigned i=0; i<myvector.size() ; i++)
  cout << " " << myvector[i];
```


Use `Iterator` to Access Element

```
vector<int> ints;
for (int i=1; i<=5; i++)
    ints.push_back(i);

vector<int>::iterator it;
cout << "my vector contains:";
for(it=ints.begin(); it<ints.end(); it++)
   cout << " " << *it;
```

```
1  // Fig. 23.14: Fig23_14.cpp
2  // Demonstrating Standard Library vector class template.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <vector> // vector class-template definition
8  using std::vector;
9
10 // prototype for function template printVector
11 template < typename T > void printVector( const vector< T > &integers2 );
12
13 int main()
14 {
15    const int SIZE = 6; // define array size
16    int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 }; // initialize array
17    vector< int > integers; // create vector of ints
18
19    cout << "The initial size of integers is: " << integers.size()
20       << "\nThe initial capacity of integers is: " << integers.capacity();
21
```

Define a **vector** called **integers** that stores **int** values

Return the number of elements that can be stored in the **vector** before it needs to dynamically resize itself to accommodate more elements

Return the number of elements currently stored in the container


```
22    // function push_back is in every sequence collection
23    integers.push_back( 2 );
24    integers.push_back( 3 );
25    integers.push_back( 4 );
26
27    cout << "\nThe size of integers is: " << integers.size()
28       << "\nThe capacity of integers is: " << integers.capacity();
29    cout << "\n\nOutput array using pointer notation: ";
30
31    // display array using pointer notation
32    for ( int *ptr = array; ptr != array + SIZE; ptr++ )
33       cout << *ptr << ' ';
34
35    cout << "\nOutput vector using iterator notation: ";
36    printVector( integers );
37    cout << "\nReversed contents of vector integers: ";
38
39    // two const reverse iterators
40    vector< int >::const_reverse_iterator reverseIterator;
41    vector< int >::const_reverse_iterator tempIterator = integers.rend();
42
43    // display vector in reverse order using reverse_iterator
44    for ( reverseIterator = integers.rbegin();
45       reverseIterator != tempIterator; ++reverseIterator )
46       cout << *reverseIterator << ' ';
47
48    cout << endl;
49    return 0;
50 } // end main
```

Add elements to the end of the **vector**

reverseIterator iterates from the position returned by **rbegin** until just before the position returned by **rend** to output the vector elements in reverse order

```
51
52 // function template for outputting vector elements
53 template < typename T > void printVector( const vector< T > &integers2 )
54 {
55    typename vector< T >::const_iterator constIterator; // const_iterator
56
57    // display vector elements using const_iterator
58    for ( constIterator = integers2.begin();
59       constIterator != integers2.end(); ++constIterator )
60       cout << *constIterator << ' ';
61 } // end function printVector
```

```
The initial size of integers is: 0
The initial capacity of integers is: 0
The size of integers is: 3
The capacity of integers is: 4

Output array using pointer notation: 1 2 3 4 5 6
Output vector using iterator notation: 2 3 4
Reversed contents of vector integers: 4 3 2
```

Tell the compiler that **vector< T > ::const_iterator** is expected to be a type in every specialization

constIterator iterates through the **vector** and outputs its contents

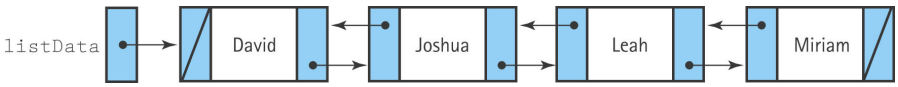
The **vector**'s capacity increases to accommodate the growing size

Applications

- Vectors are good at:
  - Accessing individual elements by their position index (constant time).
  - Iterating over the elements in any order (linear time).
  - Add and remove elements from its end (constant amortized time).

list

- List containers are implemented as doubly-linked lists;
- **Doubly linked list**
  - Each node is linked to both its successor and its predecessor



list

- Available to anyone building applications with C++

```
#include <list>
  using std::list;
```

- Can be defined to store any data type
  - Specified between angle brackets in `list<type>`
  - All elements in a `list` are set to 0 by default

```
list<int> integers;
```

`Iterator` of `list`

- Supports bidirectional iterators
  - Can be traversed forward and backward

Iterator operation	Description
`++p`	Preincrement an iterator.
`p++`	Postincrement an iterator.
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
`p == p1`	Compare iterators for equality.
`p != p1`	Compare iterators for inequality.
`--p`	Predecrement an iterator.
`p--`	Postdecrement an iterator.

Constructors

- `list<int> first;`
  `// empty list of ints`
- `list<int> second (4,100);`
  `//four ints with value 100`
- `list<int> third (second.begin(),second.end());`
  `//iterating through second`
- `list<int> fourth (third);`
  `//a copy of third`
- `int myints[] = {16,2,77,29};`

```
list<int> fifth (myints,
             myints + sizeof(myints)/sizeof(int) );
```

Member Functions of List

- Same member functions as `vector`
  - `begin, end, rbegin, rend, size, empty, front, back, push_back, pop_back, insert, erase, clear`

Comparison Function

- A comparison function that, taking two values of the same type than those contained in the list object, returns **true** if the first argument is less than the second, and **false** otherwise.

```
// this compares equal two doubles
// if their interger equivalents are equal
bool mycomparison (double first, double second)
{
   return ( int(first)< int(second) );
}
```

Member Functions

- Member function `sort`
  - Arranges the elements in the `list` in ascending order
  - Can take a binary predicate function as second argument to determine sorting order

```
list<double> values;
values.sort(mycomparison);
values.sort( );
```


Member Functions

- Member function `splice`
  - Removes elements from the container argument and inserts them into the current `list` at the specified location

```
list<double> values;
list<double> otherValues;
values.splice(values.end(), otherValues);
```

Member Functions

- Member function `merge`
  - Removes elements from the specified `list` and inserts them in **sorted order** into the current `list`
    - Both `lists` must first be sorted in the same order
    - Can take a comparison function as second argument to determine sorting order

```
values.sort();
otherValues.sort();
values.merge(otherValues);
values.merge(otherValues, mycomparison);
```


Member Function

- Member function `swap`
  - Exchange the contents of the current list with the argument.
  - Available for all containers (**`vector`**)

```
values.swap(otherValues);

vectorOne.swap(otherVector);
```

Member Function

- Member function `remove`
  - Removes from the list all the elements with a specific *value*. This calls the destructor of these objects and reduces the list size by the amount of elements removed.

```
values.remove(4.0);
```


Add & Delete Element From Front

- `void push_front(const T& ele)`
  - Inserts a new element at the beginning of the list, right before its current first element.
- `void pop_front()`
  - Removes the first element in the list container, effectively reducing the list size by one.

```
1  // Fig. 23.17: Fig23_17.cpp
2  // Standard library list class template test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <list> // list class-template definition
8  #include <algorithm> // copy algorithm
9  #include <iterator> // ostream_iterator
10
11 // prototype for function template printList
12 template < typename T > void printList( const std::list< T > &listRef );
13
14 int main()
15 {
16    const int SIZE = 4;
17    int array[ SIZE ] = { 2, 6, 4, 8 };
18    std::list< int > values; // create list of ints
19    std::list< int > otherValues; // create list of ints
20
21    // insert items in values
22    values.push_front( 1 );
23    values.push_front( 2 );
24    values.push_back( 4 );
25    values.push_back( 3 );
26
27    cout << "values contains: ";
28    printList( values );
```

Instantiate two **list** objects capable of storing integers

Insert integers at the beginning and end of **values**


```
29
30    values.sort(); // sort values
31    cout << "\nvalues after sorting contains: ";
32    printList( values );
33
34    // insert elements of array into otherValues
35    otherValues.insert( otherValues.begin(), array, array + SIZE );
36    cout << "\nAfter insert, otherValues contains: ";
37    printList( otherValues );
38
39    // remove otherValues elements and insert at end of values
40    values.splice( values.end(), otherValues );
41    cout << "\nAfter splice, values contains: ";
42    printList( values );
43
44    values.sort(); // sort values
45    cout << "\nAfter sort, values contains: ";
46    printList( values );
47
48    // insert elements of array into otherValues
49    otherValues.insert( otherValues.begin(), array, array + SIZE );
50    otherValues.sort();
51    cout << "\nAfter insert, otherValues contains: ";
52    printList( otherValues );
```

Arrange the elements in the **list** in ascending order

Remove the elements in **otherValues** and insert them at the end of **values**

```
53
54    // remove otherValues elements and insert into values in sorted order
55    values.merge( otherValues );
56    cout << "\nAfter merge:\n   values contains: ";
57    printList( values );
58    cout << "\n   otherValues contains: ";
59    printList( otherValues );
60
61    values.pop_front(); // remove element from front
62    values.pop_back(); // remove element from back
63    cout << "\nAfter pop_front and pop_back:\n   values contains: "
64    printList( values );
65
66    values.unique(); // remove duplicate elements
67    cout << "\nAfter unique, values contains: ";
68    printList( values );
69
70    // swap elements of values and otherValues
71    values.swap( otherValues );
72    cout << "\nAfter swap:\n   values contains: ";
73    printList( values );
74    cout << "\n   otherValues contains: ";
75    printList( otherValues );
76
77    // replace contents of values with elements of otherValues
78    values.assign( otherValues.begin(), otherValues.end() );
79    cout << "\nAfter assign, values contains: ";
80    printList( values );
```

Remove all elements of **otherValues** and insert them in sorted order in **values**

Remove duplicate elements in **values**

Exchange the contents of **values** with the contents of **otherValues**

Replace the contents of **values** with the elements in **otherValues**


```
81
82    // remove otherValues elements and insert into values in sorted order
83    values.merge( otherValues );
84    cout << "\nAfter merge, values contains: ";
85    printList( values );
86
87    values.remove( 4 ); // remove all 4s
88    cout << "\nAfter remove( 4 ), values contains: ";
89    printList( values );
90    cout << endl;
91    return 0;
92 } // end main
93
94 // printList function template definition; uses
95 // ostream_iterator and copy algorithm to output list elements
96 template < typename T > void printList( const std::list< T > &listRef )
97 {
98    if ( listRef.empty() ) // list is empty
99       cout << "List is empty";
100   else
101   {
102      std::ostream_iterator< T > output( cout, " " );
103      std::copy( listRef.begin(), listRef.end(), output );
104   } // end else
105} // end function printList
```

Delete all copies of the value **4** from **values**

Results

```
values contains: 2 1 4 3
values after sorting contains: 1 2 3 4
After insert, otherValues contains: 2 6 4 8
After splice, values contains: 1 2 3 4 2 6 4 8
After sort, values contains: 1 2 2 3 4 4 6 8
After insert, otherValues contains: 2 4 6 8
After merge:
   values contains: 1 2 2 2 3 4 4 4 6 6 8 8
   otherValues contains: List is empty
After pop_front and pop_back:
   values contains: 2 2 2 3 4 4 4 6 6 8
After unique, values contains: 2 3 4 6 8
After swap:
   values contains: List is empty
   otherValues contains: 2 3 4 6 8
After assign, values contains: 2 3 4 6 8
After merge, values contains: 2 2 3 3 4 4 6 6 8 8
After remove( 4 ), values contains: 2 2 3 3 6 6 8 8
```


Applications

- Advantages of double-linked list:
  - Efficient insertion and removal of elements anywhere in the container (constant time).
  - Efficient moving elements and block of elements within the container or even between different containers (constant time).
  - Iterating over the elements in forward or reverse order (linear time).

Vector vs. List

- Accessing element : vector > list
- Add or remove elements from the end of the sequence : vector > list
- Inserting or removing elements at positions other than the end: list > vector

STL associative containers

- Provide direct access to store and retrieve elements via keys (often called search keys)
  - Maintain keys in sorted order
- `iterator` that cannot be used to change element values in associative containers.

STL associative containers

- Four associative containers
  - **`multiset`** – stores keys only, allows duplicates
  - **`set`** – stores keys only, no duplicates
  - **`multimap`** – stores keys and associated values, allows duplicates
  - **`map`** – stores keys and associated values, no duplicates
- Common member functions
  - `begin, end, rbegin, rend, empty, size, swap, clear,` **`find`**, **`lower_bound`**, **`upper_bound`**,

`multiset` Associative Container

- Provides fast storage and retrieval of keys and allows duplicate keys
- The keys are always sorted in ascending order.
- Ordering of keys is determined by a comparator function object
  - Default is **`std::less< T >`** for ascending order
  - Data type of the keys must support this function
- Supports **bidirectional** iterators
- Requires header file `<set>`

multiset Associative Container

- Constructors

```
multiset<int> first; // empty multiset of ints

// pointers used as iterators
int myints[]= {10,20,30,20,20};

multiset<int> second (myints,myints+5);

multiset<int> third (second); // a copy of second

multiset<int> fourth
    (second.begin(), second.end());
```


multiset Associative Container

- Member function **insert**
  - Adds a value to a **set** or **multiset**

```
iterator insert ( const value_type& x );
//returns an iterator pointing to the newly inserted element
iterator insert ( iterator position,const value_type& x );
   //but an indication of a possible insertion position

template <class InputIterator>
void insert ( InputIterator first, InputIterator last );
//Copies of the elements in the range [first,last)
//are inserted in the multiset
```

```
multiset<int> mymultiset;
multiset<int>::iterator it;

// set some initial values:
for (int i=1; i<=5; i++)
  mymultiset.insert(i*10); // 10 20 30 40 50

it = mymultiset.insert(25);
it = mymultiset.insert (it,27); // max efficiency inserting
it = mymultiset.insert (it,29); // max efficiency inserting
it = mymultiset.insert (it,24);
                           //no max efficiency inserting (24<29)
int myints[]= {5,10,15};
mymultiset.insert (myints,myints+3);

cout << "mymultiset contains:";

for (it=mymultiset.begin(); it!=mymultiset.end(); it++)
     cout << " " << *it;

cout << endl;
```


`multiset` Associative Container

- Member function `find`
  - Locates a value in the associative container
    - Returns an `iterator` to its earliest occurrence
    - Returns the `iterator` returned by **`end`** if the value is not found

```
iterator find ( const key_type& x ) const;
```

multiset Associative Container

- Member function erase
  - Removes elements from the container.

```
void erase ( iterator position );

size_type erase ( const key_type& x );
//All the elements with this value x are removed.

void erase ( iterator first, iterator last );
//Iterators specifying a range within the
//container to be removed: [first,last).
```



```
multiset<int> mymultiset;
multiset<int>::iterator it;

for (int i=1; i<=5; i++)
    mymultiset.insert(i*10); // 10 20 30 40 50

it = mymultiset.find(20);
mymultiset.erase (it);

mymultiset.erase (mymultiset.find(40));

cout << "mymultiset contains:";

for (it=mymultiset.begin(); it!=mymultiset.end(); it++)
         cout << " " << *it;

cout << endl;
```

multiset Associative Container

- Member function lower_bound
  - Returns an **iterator** pointing to the *first* element in the container which does not compare less than the value (equal or greater).
  - Returns **end()** if no such element exists.

```
//Set is {10, 20, 30, 40, 50, 60, 70}
multiset<int>::iterator itlow;

itlow=mymultiset.lower_bound (22);
cout <<"lower_bound of 22 is "
     << *itlow <<endl;
itlow=mymultiset.lower_bound (30);
cout <<"lower_bound of 30 is "
     << *itlow <<endl;
```


multiset Associative Container

- Member function upper_bound
  - Returns an **iterator** to the *first* element in the container which compares greater than *x*.
  - Returns **end()** if no such element exists

```
//Set is {10, 20, 30, 40, 50, 60, 70}
multiset<int>::iterator itup;

itlow=mymultiset.upper_bound (22);
cout <<“upper_bound of 22 is "
     << *itup <<endl;
itlow=mymultiset.upper_bound (50);
cout <<“upper_bound of 50 is "
     << *itup <<endl;
```

```
multiset<int> mymultiset;
multiset<int>::iterator it,itlow,itup;

for (int i=1; i<8; i++)
  mymultiset.insert(i*10);        // 10 20 30 40 50 60 70

itlow=mymultiset.lower_bound (30);//       ^
itup=mymultiset.upper_bound (50); //                ^

mymultiset.erase(itlow,itup);

cout << "mymultiset contains:";

for (it=mymultiset.begin(); it!=mymultiset.end(); it++)
         cout << " " << *it;

cout << endl;
```


`multiset` associative container

- Member function `equal_range`
  - Returns a **`pair`** object containing the results of `lower_bound` and `upper_bound`
    - `pair` data member `first` stores `lower_bound`
    - `pair` data member `second` stores `upper_bound`

```
pair<iterator,iterator>
      equal_range ( const key_type& x ) const;
```

```
multiset<int>::iterator it;
pair<multiset<int>::iterator,multiset<int>::iterator> ret;

int myints[]= {77,30,16,2,30,30};
multiset<int> mymultiset (myints, myints+6);
                                  // 2 16 30 30 30 77

ret = mymultiset.equal_range(30); //      ^        ^

mymultiset.erase(ret.first, ret.second);

cout << "mymultiset contains:";

for (it=mymultiset.begin(); it!=mymultiset.end(); it++)
         cout << " " << *it;

cout << endl;
```


`set` Associative Container

- Used for fast storage and retrieval of unique keys
  - Does not allow duplicate keys
    - An attempt to insert a duplicate key is ignored
- The keys are always sorted in ascending order
- Supports bidirectional iterators
- Requires header file <`set`>

set Associative Container

- Member function insert: inserts a value into the set

```
pair<iterator, bool> insert
              ( const value_type& x );
//an iterator pointing to the element with that value,
//a bool indicating whether the value was inserted

iterator insert ( iterator position,
                           const value_type& x );
template <class Iterator>

void insert (Iterator first, Iterator last );
```



```
set<int> myset;
set<int>::iterator it;
pair<set<int>::iterator,bool> ret;

// set some initial values:
for (int i=1; i<=5; i++)
    myset.insert(i*10); // set: 10 20 30 40 50

ret = myset.insert(20); // no new element inserted

if (ret.second==false)
      it=ret.first; // "it" now points to element 20

myset.insert (it,25);
myset.insert (it,26);

int myints[]= {5,10,15}; // 10 already in set,not inserted
myset.insert (myints,myints+3);

cout << "myset contains:";
for (it=myset.begin(); it!=myset.end(); it++)
        cout << " " << *it;
```

map Associative Container

- Used for fast storage and retrieval of keys and associated values (key/value pairs)
  - Stored as std::pair objects in <utility>
  - Only one value can be associated with each key
- Commonly called an associative array
- Insertions and deletions can be made anywhere
- Requires header file <map>

map Associative Container

- Constructors

```
map<char,int> first;

first.insert(pair<char,int>('a',10));
first.insert(pair<char,int>('b',15));
first.insert(pair<char,int>('c',25));

map<char,int>second(first.begin(),first.end());

map<char,int> third (second); //copy
```

```
map<char,int> mymap;

map<char,int>::iterator it;

mymap.insert (pair<char,int>('a',10));
mymap.insert (pair<char,int>('b',20));
mymap.insert (pair<char,int>(‘c',150));

// show content:
for (it=mymap.begin() ; it != mymap.end(); it++ )
         cout << (*it).first
              << " => "
              << (*it).second << endl;
```


map Associative Container

- Subscript operator [] can locate the value associated with a given key
  - When the key is already in the map
    - Returns a reference to the associated value
  - When the key is not in the map
    - Inserts the key in the map
    - Returns a reference to the associated value (so it can be set)

```
map<char,string> mymap;
map<char,string>::iterator it;

mymap['a']="an element";
mymap['b']="another element";
mymap['c']=mymap['b'];

cout << "mymap['a'] is " << mymap['a'] << endl;
cout << "mymap['b'] is " << mymap['b'] << endl;
cout << "mymap['c'] is " << mymap['c'] << endl;
cout << "mymap['d'] is " << mymap['d'] << endl;

cout << "mymap now contains "
     << (int) mymap.size()
     << " elements." << endl;
```


`multimap` Associative Container

- Used for fast storage and retrieval of keys and associated values (key/value pairs)
  - Stored as `pair` objects
  - Duplicate keys are allowed (one-to-many mapping)
    - Multiple values can be associated with a single key
- Ordering of keys is determined by a comparator function object
- Supports bidirectional iterators
- Requires header file <map>

`multimap` Associative Container

Key	Value
John	CS2000
John	CS2200
Lisa	CS1600
Mike	CS1100

STL stack

- A stack is an adaptor that provides a restricted subset of Container functionality: it provides insertion, removal, and inspection of the element at the top of the stack. Stack is a "last in first out" (LIFO) data structure
- Stack is a container adaptor.
  - it is implemented on top of some underlying container type
- Stack does not allow iteration through its elements.

STL Stack

- Available to anyone building applications with C++

```
#include <stack>
  using std::stack;
```

- Can be defined to store any data type
  - Specified between angle brackets in **`stack<`*`type`*`>`**
  - All elements in a **`stack`** are set to 0 by default

```
stack<int> integers
```


Member Functions

- **`stack`** constructors construct a new stack
- **`empty`** true if the stack has no elements
- **`pop`** removes the top element of a stack
- **`push`** adds an element to the top of the stack
- **`size`** returns the number of items in the stack
- **`top`** returns the top element of the stack

Example

```
#include <stack>
int main()
{
    stack<int> s;
   for( int i = 0; i < 5; i++ )
      s.push(i);

   while( !s.empty() )
   {
   cout << s.top() << endl;
   s.pop();
   }
   return 0;
}
```



```
1  // Fig. 23.23: Fig23_23.cpp
2  // Standard Library adapter stack test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <stack> // stack adapter definition
8  #include <vector> // vector class-template definition
9  #include <list> // list class-template definition
10
11 // pushElements function-template prototype
12 template< typename T > void pushElements( T &stackRef );
13
14 // popElements function-template prototype
15 template< typename T > void popElements( T &stackRef );
16
17 int main()
18 {
19    // stack with default underlying deque
20    std::stack< int > intDequeStack;
21
22    // stack with underlying vector
23    std::stack< int, std::vector< int > > intVectorStack;
24
25    // stack with underlying list
26    std::stack< int, std::list< int > > intListStack;
```

Specify integer **stack**s using each of the three sequence containers as the underlying data structure

```
27
28    // push the values 0-9 onto each stack
29    cout << "Pushing onto intDequeStack: ";
30    pushElements( intDequeStack );
31    cout << "\nPushing onto intVectorStack: ";
32    pushElements( intVectorStack );
33    cout << "\nPushing onto intListStack: ";
34    pushElements( intListStack );
35    cout << endl << endl;
36
37    // display and remove elements from each stack
38    cout << "Popping from intDequeStack: ";
39    popElements( intDequeStack );
40    cout << "\nPopping from intVectorStack: ";
41    popElements( intVectorStack );
42    cout << "\nPopping from intListStack: ";
43    popElements( intListStack );
44    cout << endl;
45    return 0;
46 } // end main
47
48 // push elements onto stack object to which stackRef refers
49 template< typename T > void pushElements( T &stackRef )
50 {
51    for ( int i = 0; i < 10; i++ )
52    {
53       stackRef.push( i ); // push element onto stack
54       cout << stackRef.top() << ' '; // view (and display ) top element
55    } // end for
56 } // end function pushElements
```

Place an integer on top of the **stack**

Retrieve, but not remove, the top element


```
57
58 // pop elements from stack object to which stackRef refers
59 template< typename T > void popElements( T &stackRef )
60 {
61    while ( !stackRef.empty() )
62    {
63       cout << stackRef.top() << ' '; // view (and display) top element
64       stackRef.pop(); // remove top element
65    } // end while
66 } // end function popElements
```

Retrieve and display the top element

Remove, and discard, the top element

```
Pushing onto intDequeStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto intVectorStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto intListStack: 0 1 2 3 4 5 6 7 8 9

Popping from intDequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from intVectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from intListStack: 9 8 7 6 5 4 3 2 1 0
```

Class queue

- Enables insertions at back and deletions from front
  - First-in, first-out data structure
- Can be implemented with data structure `list` or `deque`
  - Implemented with a `deque` by default
- Requires header file `<queue>`

Class queue

- Operations (call functions of the underlying container)
  - `push` – insert element at back (calls `push_back`)
  - `pop` – remove element from front (calls `pop_front`)
  - `front` – returns reference to first element (calls `front`)
  - `empty` – determine if the `queue` is empty (calls `empty`)
  - `size` – get the number of elements (calls `size`)

```
1  // Fig. 23.24: Fig23_24.cpp
2  // Standard Library adapter queue test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <queue> // queue adapter definition
8
9  int main()
10 {
11    std::queue< double > values; // queue with doubles
12
13    // push elements onto queue values
14    values.push( 3.2 );
15    values.push( 9.8 );
16    values.push( 5.4 );
17
18    cout << "Popping from values: ";
```

Instantiate a **queue** that stores **double** values

Add elements to the **queue**


```
19
20    // pop elements from queue
21    while ( !values.empty() )
22    {
23       cout << values.front() << ' '; // view front element
24       values.pop(); // remove element
25    } // end while
26
27    cout << endl;
28    return 0;
29 } // end main
```

Read the first element in the **queue** for output

Remove the first element in the **queue**

```
Popping from values: 3.2 9.8 5.4
```

STL Internet and Web Resources

- Tutorials
  - www.cs.brown.edu/people/jak/programming/stl-tutorial/tutorial.html
    - STL tutorial organized by examples, philosophy, components and extending the STL
  - www.yrl.co.uk/phil/stl/stl.htmlx
    - Function templates, class templates, STL components, containers, iterators, adaptors and function objects
  - www.xraylith.wisc.edu/~khan/software/stl/os_examples/examples.html
    - Introduction to STL and ObjectSpace STL Tool Kit

STL Internet and Web Resources

- References
  - www.sgi.com/tech/stl
    - Silicon Graphics STL Programmer's Guide
      - Latest information, design documentation and links
  - www.cppreference.com/cpp_stl.html
    - Lists constructors, operators and functions for each container
- Articles, Books and Interviews
  - www.byte.com/art/9510/sec12/art3.htm
    - Provides information on the use of STL
  - www.sgi.com/tech/stl/drdobbs-interview.html
    - Interview with Alexander Stepanov, one of the STL creators

STL Internet and Web Resources

- ANSI/ISO C++ Standard
  - www.ansi.org
    - C++ standard document available for purchase
- Software
  - www.cs.rpi.edu/~musser/stl-book
    - Information and resources for using STL
  - msdn.microsoft.com/visualc
    - Microsoft Visual C++ home page – news, updates, technical resources, samples and downloads
  - www.borland.com/cbuilder
    - Borland C++Builder home page – newsgroups, product enhancements, FAQs and more

Reference

- Reproduced from the Cyber Classroom for C++, How to Program**, 5/e by Deitel & Deitel.**
- **Reproduced by permission of Pearson Education, Inc.**