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 Template Library

• The Standard Library is a fundamental part of the C++ Standard.
  – a comprehensive set of efficiently implemented tools and facilities.
• Standard Template Library (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

• Three major categories
  – Sequence containers
    • Represent linear data structures
  – Associative containers
    • Nonlinear containers
    • Store key/value pairs
  – Container adapters
    • Implemented as constrained sequence containers

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

<table>
<thead>
<tr>
<th>Predefined typedefs for iterator types</th>
<th>Direction of ++</th>
<th>Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iterator</code></td>
<td>forward</td>
<td>read/write</td>
</tr>
<tr>
<td><code>const_iterator</code></td>
<td>forward</td>
<td>read</td>
</tr>
<tr>
<td><code>reverse_iterator</code></td>
<td>backward</td>
<td>read/write</td>
</tr>
<tr>
<td><code>const_reverse_iterator</code></td>
<td>backward</td>
<td>read</td>
</tr>
</tbody>
</table>

Fig. 23.9 | Iterator typedefs.
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

- Vectors are a kind of sequence containers. As such, their 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++

```cpp
#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

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

Iterator of Vector

• A vector supports random-access iterators.
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();
  ```

- 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.
  
  ```cpp
  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.
  
  ```cpp
  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()`

```
    1  2  3  4  5  7  8
rend()     begin()     rbegin()     end()
```
<table>
<thead>
<tr>
<th>Iterator operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>++p</code></td>
<td>Preincrement an iterator.</td>
</tr>
<tr>
<td><code>p++</code></td>
<td>Postincrement an iterator.</td>
</tr>
<tr>
<td><code>*p</code></td>
<td>Dereference an iterator.</td>
</tr>
<tr>
<td><code>p = p1</code></td>
<td>Assign one iterator to another.</td>
</tr>
<tr>
<td><code>p == p1</code></td>
<td>Compare iterators for equality.</td>
</tr>
<tr>
<td><code>p != p1</code></td>
<td>Compare iterators for inequality.</td>
</tr>
<tr>
<td><code>--p</code></td>
<td>Predecrement an iterator.</td>
</tr>
<tr>
<td><code>p--</code></td>
<td>Postdecrement an iterator.</td>
</tr>
</tbody>
</table>

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

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.
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

```cpp
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

```cpp
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;
```
```cpp
// Fig. 23.14: Fig23_14.cpp
// Demonstrating Standard Library vector class template.
#include <iostream>
using std::cout;
using std::endl;
#include <vector> // vector class - template definition
using std::vector;

// prototype for function template printVector
template <typename T>
void printVector(const vector<T> &integers2);

int main()
{
    const int SIZE = 6; // define array size
    int array[SIZE] = {1, 2, 3, 4, 5, 6}; // initialize array
    vector<int> integers; // create vector of ints

    cout << "The initial size of integers is: " << integers.size() << endl;
    cout << "The initial capacity of integers is: " << integers.capacity() << endl;

    integers.push_back(2);
    integers.push_back(3);
    integers.push_back(4);

    cout << "The size of integers is: " << integers.size() << endl;
    cout << "The capacity of integers is: " << integers.capacity() << endl;

    cout << "Output array using pointer notation:
    for (int *ptr = array; ptr != array + SIZE; ptr++)
    cout << *ptr << ' ';

    cout << endl;

    cout << "Output vector using iterator notation:
    printVector(integers);
    cout << "Reversed contents of vector integers: ";
    // two const reverse iterators
    vector<int>::const_reverse_iterator reverseIterator;
    vector<int>::const_reverse_iterator tempIterator = integers.rbegin();
    // display vector in reverse order using reverse_iterator
    for (reverseIterator = integers.rbegin(); reverseIterator != tempIterator; ++reverseIterator)
    cout << *reverseIterator << ' ';
    cout << endl;
    return 0;
} // end main
```

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

Add elements to the end of the `vector`

Return the number of elements currently stored in the container

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

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
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

\[
\text{listData} \quad \rightarrow \quad \text{David} \quad \rightarrow \quad \text{Joshua} \quad \rightarrow \quad \text{Leah} \quad \rightarrow \quad \text{Miriam}
\]

list

- Available to anyone building applications with C++
  
  ```cpp
  #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

<table>
<thead>
<tr>
<th>Iterator operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>++p</td>
<td>Preincrement an iterator.</td>
</tr>
<tr>
<td>p++</td>
<td>Postincrement an iterator.</td>
</tr>
<tr>
<td>*p</td>
<td>Dereference an iterator.</td>
</tr>
<tr>
<td>p = p1</td>
<td>Assign one iterator to another.</td>
</tr>
<tr>
<td>p == p1</td>
<td>Compare iterators for equality.</td>
</tr>
<tr>
<td>p != p1</td>
<td>Compare iterators for inequality.</td>
</tr>
<tr>
<td>--p</td>
<td>Predecrement an iterator.</td>
</tr>
<tr>
<td>p--</td>
<td>Postdecrement an iterator.</td>
</tr>
</tbody>
</table>

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

```cpp
list<double> values;
values.sort(mycmp);
values.sort();
```

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.
// Fig. 23.17: Fig23_17.cpp
// Standard library list class template test program.
#include <iostream>
using std::cout;
using std::endl;

#include <list> // list class - template definition
#include <algorithm> // copy algorithm
#include <iterator> // ostream_iterator

// prototype for function template printList
template <typename T>
void printList(const std::list<T> &listRef);

int main()
{
    const int SIZE = 4;
    int array[SIZE] = {2, 6, 4, 8};
    std::list<int> values; // create list of ints
    std::list<int> otherValues; // create list of ints

    // insert items into values
    values.push_front(1);
    values.push_front(2);
    values.push_back(4);
    values.push_back(3);

    cout << "values contains: ";
    printList(values);

    values.sort(); // sort values
    cout << "\nvalues after sorting contains: ";
    printList(values);

    // insert elements of array into otherValues
    otherValues.insert(otherValues.begin(), array, array + SIZE);
    cout << "\nAfter insert, otherValues contains: ";
    printList(otherValues);

    // remove otherValues elements and insert at end of values
    values.splice(values.end(), otherValues);
    cout << "\nAfter splice, values contains: ";
    printList(values);
    values.sort(); // sort values
    cout << "\nAfter sort, values contains: ";
    printList(values);

    // insert elements of array into otherValues
    otherValues.insert(otherValues.begin(), array, array + SIZE);
    otherValues.sort();
    cout << "\nAfter insert, otherValues contains: ";
    printList(otherValues);
}

// instantiates two list objects capable of storing integers
// insert integers at the beginning and end of values
// arrange the elements in the list in ascending order
// remove the elements in otherValues and insert them at the end of values
// remove otherValues elements and insert into values in sorted order
values.merge(otherValues);
// remove all 4s from values
values.remove(4);
// printList function template definition; uses
// ostream_iterator and copy algorithm to output list elements
template<typename T>
void printList(const std::list<T> &listRef)
{
    if (listRef.empty()) // list is empty
        cout << "List is empty";
    else
    {
        std::ostream_iterator<T> output(cout, " ");
        std::copy(listRef.begin(), listRef.end(), output);
    }
} // end function printList
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 3 4 4 6 8
After insert, otherValues contains: 2 4 6 8
After merge:
   values contains: 1 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 3 4 4 4 6 6 8
   otherValues contains: List is empty
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 4 4 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

\[
\text{iterator find ( const key_type& x ) const;}
\]
multiset Associative Container

• Member function erase
  – Removes elements from the container.

  ```cpp
  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;

```cpp
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;
```
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`

Member function `insert`: inserts a value into the set

```cpp
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
```

```cpp
iterator insert ( iterator position,
                 const value_type& x );
```

```cpp
template <class Iterator>
void insert (Iterator first, Iterator last );
```
```cpp
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 pair objects
  - 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

```cpp
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
```

```cpp
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)

```cpp
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>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>CS2000</td>
</tr>
<tr>
<td>John</td>
<td>CS2200</td>
</tr>
<tr>
<td>Lisa</td>
<td>CS1600</td>
</tr>
<tr>
<td>Mike</td>
<td>CS1100</td>
</tr>
</tbody>
</table>
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

```
#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;
}
```
// Fig. 23.23: Fig23_23.cpp
// Standard Library adapter stack test program.
#include <iostream>
#include <stack>
#include <vector>
#include <list>

// pushElements function-template prototype
template<typename T> void pushElements( T &stackRef );

// popElements function-template prototype
template<typename T> void popElements( T &stackRef );

int main()
{
    // stack with default underlying deque
    std::stack<int> intDequeStack;
    // stack with underlying vector
    std::stack<int, std::vector<int>> intVectorStack;
    // stack with underlying list
    std::stack<int, std::list<int>> intListStack;

    // push the values 0-9 onto each stack
    cout << "Pushing onto intDequeStack: " ;
    pushElements( intDequeStack );
    cout << "Pushing onto intVectorStack: " ;
    pushElements( intVectorStack );
    cout << "Pushing onto intListStack: " ;
    pushElements( intListStack );

    // display and remove elements from each stack
    cout << "Popping from intDequeStack: " ;
    popElements( intDequeStack );
    cout << "Popping from intVectorStack: " ;
    popElements( intVectorStack );
    cout << "Popping from intListStack: " ;
    popElements( intListStack );

    cout << endl << endl;

    // push elements onto stack object to which stackRef refers
    template<typename T> void pushElements( T &stackRef )
    {
        for ( int i = 0; i < 10; ++i )
        {
            stackRef.push( i ); // push element onto stack
            cout << stackRef.top() << " " ; // view (and display) top element
        }
    }
    // end function pushElements

    return 0;
} // end main

// push elements onto stack object to which stackRef refers
template<typename T> void pushElements( T &stackRef )
{
    for ( int i = 0; i < 10; ++i )
    {
        stackRef.push( i ); // push element onto stack
        cout << stackRef.top() << " " ; // view (and display) top element
    }
} // end for
// end function pushElements

Specify integer stacks using each of the three sequence containers as the underlying data structure.

Place an integer on top of the stack

Retrieve, but not remove, the top element
template< typename T >
void popElements( T &stackRef )
{
    while ( !stackRef.empty() )
    {
        cout << stackRef.top() << ' ';
        // view (and display) top element
        stackRef.pop();  // remove top element
    }  // end while
}  // end function popElements

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)

```cpp
#include <iostream>
#include <queue>

int main()
{
    std::queue<double> values; // queue with doubles
    values.push(3.2);
    values.push(9.8);
    values.push(5.4);

    cout << "Popping from values: ";
```

Instantiate a queue that stores double values
Add elements to the queue

Algorithms

- The STL provides a number of useful, generic algorithms to perform the most commonly used operations on groups/sequences of elements.
  - traversals,
  - searching,
  - Sorting
  - insertion/removal of elements.
Example: find() Function

- A generic function which searches a container within a search range.
- \texttt{template <class Iterator, class T> Iterator find (Iterator first, Iterator last, const T & value);}

Example: sort() Function

- A generic function which sorts elements in the container within a range into ascending order.
- \texttt{template <class RandomAccessIterator> void sort (RandomAccessIterator first, RandomAccessIterator last);}
STL Internet and Web Resources

• Tutorials
    • STL tutorial organized by examples, philosophy, components and extending the STL
  – [www.yrl.co.uk/phil/stl/stl.htmlx](http://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](http://www.xraylith.wisc.edu/~khan/software/stl/os_examples/examples.html)
    • Introduction to STL and ObjectSpace STL Tool Kit

• References
    • Silicon Graphics STL Programmer’s Guide
      – Latest information, design documentation and links
    • Lists constructors, operators and functions for each container

• Articles, Books and Interviews
    • Provides information on the use of STL
    • 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
    • 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.