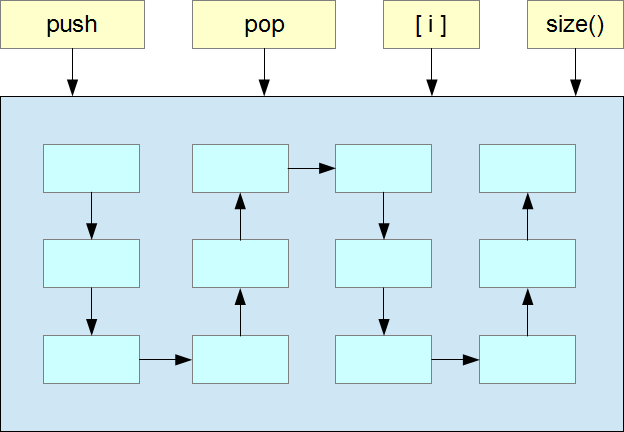
Containers and Iterators

Object-oriented languages include library support for containers of object of fundamental, built-in and user-defined types.  The container classes and container adaptors of the STL provide shells for elemental collections of types.  The sequential container classes order their elements in sequence.  The container adapters provide interfaces for encapsulating access to elements of specific containers.  The iterators of the STL provide access for iterating on the collections.  These iterators provide the glue that enables use of the STL algorithms with these collections.



This chapter introduces several sequential container classes along with their member functions.  This chapter also introduces the concept of iterators and presents examples of their use with sequential container classes.  The algorithms of the STL are described in the following chapter.

Sequential Containers

The STL defines the sequential container classes in the form of class templates.  These templates include:

* array - contiguous storage of fixed size
* vector - contiguous storage of variable size
* deque - non-contiguous storage of variable size, double-ended queue
* forward\_list - non-contiguous storage of variable size, singly linked list
* list - non-contiguous storage of variable size, doubly linked list

An array, unlike a built-in array knows its size and can be copied and assigned, but is allocated on the stack rather than on free store, so its size cannot be changed.

vector

The vector class template defines classes that manage data structures that can change in size but have their elements ordered in sequence.  These classes use contiguous storage locations for their elements and are nearly as efficient as arrays.  They store their elements on free store and can adjust their size as required.  The class template is optimized for fast random access as well as addition and deletion of elements at the back-end of a sequence.

vector

The vector class template is defined in the following header file:

#include <vector>

A vector object admits comparison, assignment, expansion, element addition, and element deletion.  Its member functions include:

* vector() - default constructor - creates a container with no elements
* vector(int n) - creates a container with n elements
* vector(int n, const T& t) - creates a container with n elements, each initialized to value t
* vector(const vector& v) - copies the contents of container v into the current object
* vector(vector&& v) noexcept - moves the contents of container v into the current object
* ~vector() - destroys the container
* vector& operator=(const vector& v) - copies the contents of container v into the current object
* vector& operator=(vector&& v) noexcept - moves the contents of container v into the current object
* size\_t size() const - returns the number of elements in the current object
* size\_t capacity() const - returns the current capacity of the current object
* bool empty() const - returns true if the current object has no elements
* T& operator[](size\_t i) - returns a reference to element i
* const T& operator[](size\_t i) const - returns an unmodifiable reference to element i
* T& at(size\_t i) - returns a reference to element i and checks bounds - throwing an exception
* const T& at(size\_t i) const - returns an unmodifiable reference to element i and checks bounds - throwing an exception
* T\* data() noexcept - returns a pointer to the underlying array
* const T\* data() const noexcept - returns a pointer to the underlying unmodifiable array
* T& front() - returns a reference to the first element
* const T& front() const - returns an unmodifiable reference to the first element
* T& back() - returns a reference to the last element
* const T& back() const - returns an unmodifiable reference to the last element
* void push\_back(const T& t) - adds element t after the last element in the container
* void pop\_back() - removes the last element from the container
* void clear() - removes all elements from the container

The two at(int) member functions check that the index received is within bounds, while the subscripting operator does not check bounds for improved efficiency.

The following program creates an initially empty sequence of doubles named prices, stores three elements, changes the first element, and discards the last element:

|  |  |
| --- | --- |
| // Sequential Containers - Vectors  // vector.cpp  #include <vector>  #include <iostream>  int main() {  std::vector<double> prices; // initially empty  if(prices.empty()) // is prices empty?  std::cout << "prices is empty" << std::endl;  prices.push\_back(10.43); // add 10.43  prices.push\_back(20.54); // add 20.54  prices.push\_back(32.43); // add 32.43  for(int i = 0; i < prices.size(); i++)  std::cout << prices[i] << " ";  std::cout << std::endl;  prices.front() = 54.11; // change 1st element  prices.pop\_back(); // remove last element  for(int i = 0; i < prices.size(); i++)  std::cout << prices[i] << " ";  std::cout << std::endl;  } | prices is empty  10.43 20.54 32.43  54.11 20.54 |

Since a vector object stores its elements contiguously, accessing the address of its first element returns a pointer to its contents.  To get the pointer to the first element of object v, we can use either &v[0] or &v.front().

The iterators described below provide an alternative method of access to ranges of elements within a vector container.

deque

The deque class template defines classes that manage doubly-ended queues that can change in size from either end and have elements ordered in sequence.  Insertion and deletion of elements at either end is efficient, but elements can be scattered throughout memory in different chunks of storage and contiguous storage is not guaranteed.

deque

The deque class template is defined in the following header file:

#include <deque>

A deque object admits comparison, assignment, expansion, element addition, and element deletion.  Its member functions include:

* deque() - default constructor - creates an empty container
* deque(int n) - creates a container with n elements
* deque(int n, const T& d) - creates a container with n elements, each initialized to value d
* deque(const deque& d) - copies the contents of d into the current object
* deque(deque&& d) noexcept - moves the contents of d into the current object
* ~deque() - destroys the container
* deque& operator=(const deque& d) - copies the contents of container d into the current object
* deque& operator=(deque&& d) noexcept - moves the contents of container d into the current object
* size\_t size() const - returns the number of elements in the current object
* size\_t capacity() const - returns the current capacity of the current object
* bool empty() const - returns true if the current object has no elements
* T& operator[](size\_t i) - returns a reference to element i
* const T& operator[](size\_t i) const - returns an unmodifiable reference to element i
* T& at(size\_t i) - returns a reference to element i, checks bounds - throwing an exception
* const T& at(size\_t i) const - returns an unmodifiable reference to element i, checks bounds - throwing an exception
* T& front() - returns a reference to the first element
* const T& front() const - returns an unmodifiable reference to the first element
* T& back() - returns a reference to the last element
* const T& back() const - returns an unmodifiable reference to the last element
* void push\_back(const T& t) - adds element t after the last element in the container
* void push\_front(const T& t) - adds element t before the first element in the container
* void pop\_back() - removes the last element from the container
* void pop\_front() - removes the first element from the container
* void clear() - removes all elements from the container

The two at(int) member functions check that the index received is within bounds, while the subscripting operator does not check bounds for improved efficiency.  The highlighted functions provide front end access and are absent in the vector class template.

The following program creates a double-ended queue of doubles named prices and initializes each element to 10.50, changes the last element's value to 32.43, removes the first element, assigns the queue to another queue, adds 5.64 and 20.31 to the front of that queue, and adds 10 to its second element:

|  |  |
| --- | --- |
| // Sequential Containers - Double-Ended Queues  // deque.cpp  #include <deque> // for deque template  #include <iostream>  int main() {  std::deque<double> prices(3, 10.50), costs;  prices.back() = 32.43; // reset last  prices.pop\_front(); // remove first  for(int i = 0; i < prices.size(); i++)  std::cout << prices[i] << " ";  std::cout << std::endl;  costs = prices;  costs.push\_front(5.64); // add 5.64  costs.push\_front(20.31); // add 20.31  costs.at(1) += 10.0; // add 10.0  for(int i = 0; i < costs.size(); i++)  std::cout << costs[i] << " ";  std::cout << std::endl;  } | 10.5 32.43  20.31 15.64 10.5 32.43 |

Unlike a vector, offsetting a pointer to another deque element causes undefined behavior. The iterators described below provide an alternative method of access for ranges of elements within a deque container.

list

The list class template defines classes that manage doubly linked sequences that are optimized for insertion and removal of elements anywhere throughout the list, particularily in larger sequences.  This template is sub-optimal for fast random access.  For fast random access, the vector and deque templates are more efficient.

The list class template is defined in the following header file:

#include <list>

A list object admits comparison, assignment, expansion, element addition, and element deletion.  Its member functions include:

* list() - default constructor - creates an empty container
* list(int n) - creates a container with n elements
* list(int n, const T& l) - creates a container with n elements, each initialized to value l
* list(const list& l) - copies the contents of v into the current object
* list(list&& l) noexcept - copies the contents of v into the current object
* ~list() - destroys the container
* list& operator=(const list& l) - copies the contents of l into the current object
* list& operator=(list&& l) noexcept - moves the contents of l into the current object
* size\_t size() const - returns the number of elements in the current object
* bool empty() const - returns true if the current object has no elements
* T& front() - returns a reference to the first element
* const T& front() const - returns an unmodifiable reference to the first element
* T& back() - returns a reference to the last element
* const T& back() const - returns an unmodifiable reference to the last element
* void push\_back(const T& t) - adds element t after the last element in the container
* void push\_front(const T& t) - adds element t before the first element in the container
* void pop\_back() - removes the last element from the container
* void pop\_front() - removes the first element from the container
* iterator insert(iterator position, const T& t) - adds element T at the iterator position
* iterator erase(iterator position, const T& t) - remove element T at the iterator position
* void clear() - removes all elements from the container

The subscripting operators and the at(int) member functions, which provide direct element access in other sequential containers, are not supported in this template.  Instead, the template define insert() and erase() member functions that use iterators (highlighted here).  The section on iterators below described these in more detail.

Container Adapters

The STL inludes adapters for converting a container class to operate in a specific context.  The adaptors include:

* stack - last in, first out (LIFO) context
* queue - first in, first out (FIFO) context
* priority\_queue - first element is always the greatest

stack

The stack class template defines a container class (by default deque) that operates in a FILO (first-in, last-out) context.  The stack class template is defined in the following header file:

#include <stack>

The member functions of the stack class template include:

* explicit stack() - default - creates a stack with no elements
* explicit stack(const Container& c) - creates a stack initialized to a copy of container c
* stack& operator=(const stack& s) - copies the contents of s into the current object
* ~stack() - destroys the stack
* size\_t size() const - returns the number of elements in the current object
* bool empty() const - returns true if the current object has no elements
* T& top() - returns a reference to the top element of the stack
* const T& top() const - returns an unmodifiable reference to the top element of the stack
* void push(const T& t) - adds element t to the top of the stack
* void pop() - removes the top element from the stack

The following program creates an initially empty stack of doubles named prices, pushes three elements onto the stack, changes the top element, and displays the elements as it pops them off the stack:

|  |  |
| --- | --- |
| // Container Adapters - Stacks  // ca\_stack.cpp  #include <stack>  #include <iostream>  int main() {  std::stack<double> prices; // initially empty  prices.push(10.43); // add 10.43  prices.push(20.54); // add 20.54  prices.push(32.43); // add 32.43  prices.top() = 5.41;  while(!prices.empty()) {  std::cout << prices.top() << " ";  prices.pop();  }  std::cout << std::endl;  } | 5.41 20.54 10.43 |

queue

The queue class template defines a container class (by default deque) that operates in a FIFO context.  The queue class template is defined in the following header file:

#include <queue>

The member functions of the queue class template include:

* explicit queue() - default constructor - creates a queue with no elements
* explicit queue(const Container& c) - creates a queue initialized to a copy of container c
* size\_t size() const - returns the number of elements in the current object
* bool empty() const - returns true if the current object has no elements
* void push(const T& t) - adds element t to the top of the queue
* void pop() - removes the first element from the queue
* T& front() - returns a reference to the first element of the queue
* const T& front() const - returns an unmodifiable reference to the first element of the queue
* T& back() - returns a reference to the last element of the queue
* const T& back() const - returns an unmodifiable reference to the last element of the queue

Note that the last 4 member functions are absent in the stack template.

The following program creates an initially empty queue of ints named tickets, pushes three elements into the queue, changes the last ticket, and displays the elements as it pops them off the queue:

|  |  |
| --- | --- |
| // Container Adapters - Queues  // ca\_queue.cpp  #include <queue>  #include <iostream>  int main() {  std::queue<int> tickets; // initially empty  tickets.push(10); // add 10  tickets.push(20); // add 20  tickets.push(32); // add 32  tickets.back() = 30;  while(!tickets.empty()) {  std::cout << tickets.front() << " ";  tickets.pop();  }  std::cout << std::endl;  } | 10 20 30 |

Iterators

An *iterator* is an object that points to an element in a sequence.  STL iterators simulate sequential access to elements of STL containers, similar to the access that raw pointers provide for elements of simple arrays.  Container classes that do not implement contiguous storage of elements require iterators to access their elements.  We use iterators to insert elements into a sequence or to remove them from a sequence.

The definition of an iterator takes the form

*Container*<*type*>::iterator *identifier*;

*Container* identifies the template of the container class, *type* is the template argument, and *identifier* is the name of the iterator.

For example, to define an iterator named iter for a vector of doubles, we write

|  |
| --- |
| std::vector<double>::iterator iter; |

Each STL container class includes the following member functions that return iterators:

* iterator begin() noexcept - returns an iterator pointing to the first element in a sequence
* iterator end() noexcept - returns an iterator pointing to the element one past the end of a sequence
* const\_iterator cbegin() const noexcept - returns an iterator pointing to the first element unmodifiable in a sequence
* const\_iterator cend() noexcept - returns an iterator pointing to the element unmodifiable one past the end of a sequence

Incrementing an iterator (++) updates it to point to the next element.  Decrementing an iterator (--) updates it to point to the previous element.  Applying the dereferencing operator (\*) to an iterator returns the value of the element pointed to by the iterator.

For example, to display the contents of a vector sequence, we write

|  |  |
| --- | --- |
| // Iterators - Vectors  // iterator.cpp  #include <vector>  #include <iostream>  int main() {  std::vector<double> prices; // initially empty  std::vector<double>::iterator i;  prices.push\_back(10.43); // add 10.43  prices.push\_back(20.54); // add 20.54  prices.push\_back(32.43); // add 32.43  for(i = prices.begin(); i != prices.end(); i++)  std::cout << \*i << " ";  std::cout << std::endl;  } | 10.43 20.54 32.43 |

We can omit the declaration of i and use auto in the initializer of the for iteration.

Inserting or Removing Elements

The container class templates define member functions for inserting and removing elements anywhere within a collection:

* iterator insert(iterator position, const T& t) - inserts t at position p and returns an iterator pointing to the inserted element
* void insert(iterator position, size\_t n, const T& t) - inserts t n times at position p
* void insert(iterator position, InIter f, InIter l) - inserts the range (f,l) at position p
* iterator erase(iterator p) - removes the element at position p and returns an iterator to the next element
* iterator erase(iterator f, iterator l) - removes the elements in the range (f,l) and returns an iterator to the next element

list Example

The following program creates an empty list, adds 3 doubles, inserts a price of 12.52 before the last element, removes the second element in the list and displays the elements in the list:

|  |  |
| --- | --- |
| // Iterators - Insertion and Removal  // iterator\_list.cpp  #include <list>  #include <iostream>  int main() {  std::list<double> prices; // initially empty  prices.push\_back(10.43); // add 10.43  prices.push\_back(20.54); // add 20.54  prices.push\_back(32.43); // add 32.43  prices.insert(--prices.end(), 12.52);  prices.erase(++prices.begin());  for(auto i = prices.begin(); i != prices.end(); i++)  std::cout << \*i << " ";  std::cout << std::endl;  } | 10.43 12.52 32.43 |

deque Example

The following program creates a deque of 3 doubles initialized to 10.50, changes the value of the last element to 32.43, removes the first element, adds 15.64 after the first element, adds 20.31 to the end and displays the elements in the list:

|  |  |
| --- | --- |
| // Iterators - Insertion and Removal  // iterator\_deque.cpp  #include <deque>  #include <iostream>  int main() {  std::deque<double> p(3, 10.50);  p.back() = 32.43; // reset last  p.erase(p.begin()); // remove first  for(auto i = p.begin(); i != p.end(); i++)  std::cout << \*i << " ";  std::cout << std::endl;  p.insert(++p.begin(), 15.64);  p.insert(p.end(), 20.31);  for(auto i = p.begin(); i != p.end(); i++)  std::cout << \*i << " ";  std::cout << std::endl;  } | 10.5 32.43  10.5 15.64 32.43 20.31 |