

# The C++ STL

* In 1990, Alex Stepanov and Meng Lee of Hewlett Packard Laboratories extended C++ with a library of class and function templates which has come to be known as the STL.
* In 1994, STL was adopted as part of ANSI/ISO Standard C++.

# Components of the STL

* Program’s main objective is to manipulate data and generate results

– Requires ability to **store** data, **access** data, and **manipulate** data

* STL has three basic components:
  1. **Containers**: generic class templates for **storing** collection of data (contain other objects).
  2. **Iterators:** generalized ‘smart’ **pointers** that provides operations for indirect access and facilitate use of containers. They provide an interface that is needed for STL algorithms to operate on STL containers.
  3. **Algorithms**: generic **function templates** for operating on containers.

Why use STL?

* STL offers an assortment of **containers**
* STL publicizes the time and storage **complexity** of its containers
* STL containers grow and shrink in **size** automatically
* STL provides built-in **algorithms** for processing containers
* STL provides **iterators** that make the containers and algorithms flexible and efficient.
* STL is **extendable** which means that users can add new containers and new algorithms.
* **Memory management**: no memory leaks or serious memoryaccess violations. (e.g., pointers)
* Reduce testing and debugging **time**.

# Sequence Containers

* Every object has a specific position
* Predefined sequence containers

–vector , deque , list

* Sequence container vector
  + Logically: same as **arrays**
* All containers
  + Use same names for common operations
  + Have specific operations

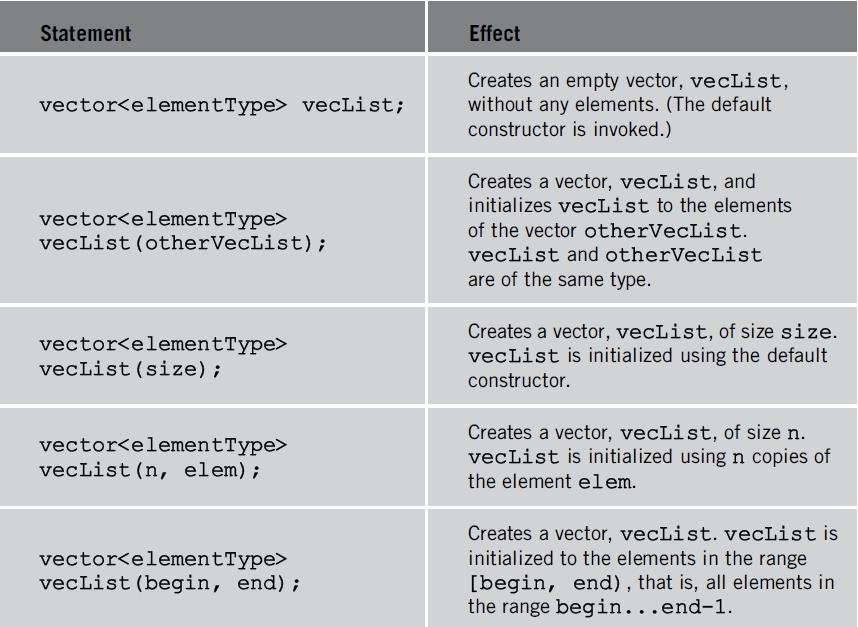
# Sequence Container: vector

* Vector container
  + Stores, manages objects in a **dynamic array**
  + Elements accessed **randomly**
  + Time-consuming item insertion: beginning and middle
  + Fast item insertion: end
* Class implementing vector container
  + vector
* Header file containing the class vector

## – vector

* Using a vector container in a program requires the following statement:
  + #include <vector>
* Declaring vector objects

Various ways to declare and initialize a vector container

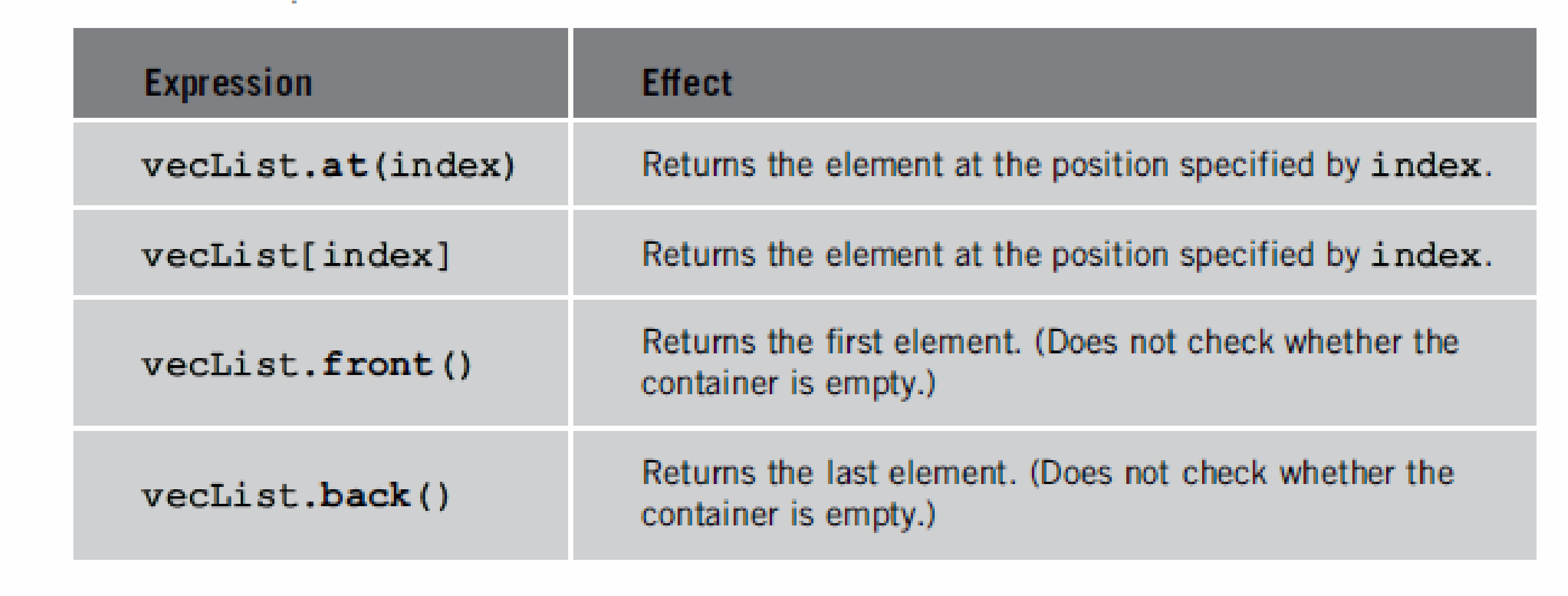


* + Examples:

•vector<int> intlist;

•vector<string> stringList;

## Operations to **access** the elements of a vector container



|  |
| --- |
| myvector contains: 0 1 2 3 4 5 6 7 8 9 |

#include <iostream> #include <vector>

int main()

{ std::vector<int> myvector(10); // 10 zero-initialized ints

// assign some values: for (unsigned i = 0; i<myvector.size(); i++) myvector.at(i) = i;

std::cout << "myvector contains:"; for (unsigned i = 0; i<myvector.size(); i++) std::cout << ' ' << myvector.at(i); std::cout << '\n'; return 0;}

# Declaring an Iterator to a Vector Container

* Process vector container like an array

– Using array subscripting operator

* Process vector container elements – Using an iterator

## • class vector: function insert

– Insert element at a specific vector container position – Uses an iterator

## • class vector: function erase

– Remove element

• Uses an iterator

### • class vector contains typedef iterator

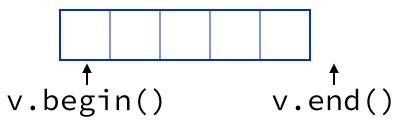
* Declared as a public member
* Vector container iterator
* Example **vector<int>::iterator intVecIter;**
* Requirements for using typedef iterator
  1. Container name (vector)
  2. Container element type (<int>)
  3. Scope resolution operator (::)
* **++intVecIter**
  + Advances iterator intVecIter to next element into the container
* **\*intVecIter**
  + Dereferencing
  + Returns element at current iterator position

Containers and the Functions begin and end

* A sequence is defined by a pair of iterators defining a half-open range **[begin:end)** 
  + Includes first element but excludes last element.
* **begin**
  + Returns an iterator to the first element in the container

### • **end**

– Returns an iterator to the element past the end. It does not point to any element. Never read from or write to end.



|  |
| --- |
| 2 2 4 2 2  4 4 6 2 4  7 7 4 2 7 |

#include <iostream> #include <vector> using namespace std; int main() { vector<int> v1; v1.push\_back(2); v1.push\_back(4); v1.push\_back(7); vector<int> v2(v1); vector<int> v3(3); v3.at(0) = 4; v3.at(1) = 6; v3.at(2) = 4; vector<int> v4(4, 2); vector<int> v5(v2.begin(), v2.end());

for (unsigned i = 0; i < v1.size(); i++)

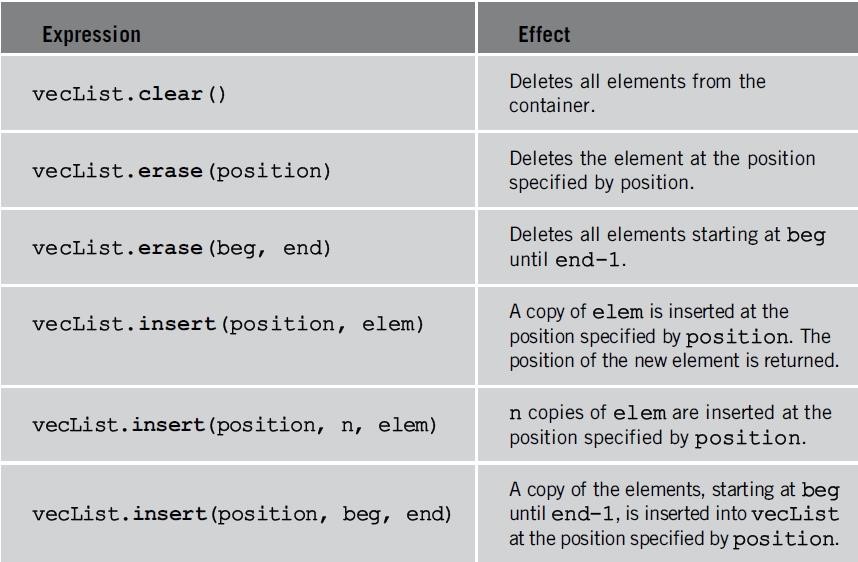
{cout << ' ' << v1.at(i) << "\t" << v2[i] << "\t" << v3.at(i) << "\t" << v4.at(i) << "\t"<< v5.at(i); cout << '\n';} return 0;}

|  |
| --- |
| 36  346 |

#include <iostream> #include <vector> using namespace std; int main() { vector<int> v1; v1.push\_back(3); v1.push\_back(4); v1.push\_back(6); vector<int>::iterator it; cout << v1.front() << v1.back() << "\n";

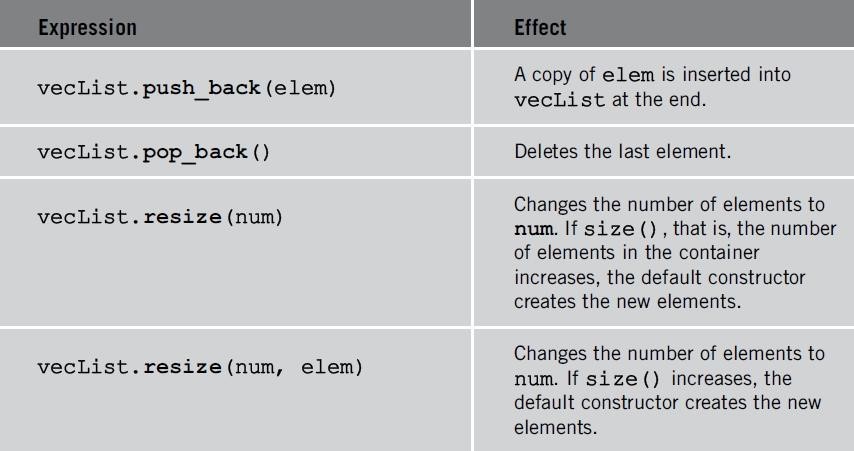
for (it = v1.begin(); it != v1.end(); it++) cout << \*it; return 0;}

## Various operations on a vector container



### • **position is an iterator**

* **insert():** the vector is extended by inserting new elements before the element at the specified position, effectively increasing the container size by the number of elements inserted.
* **Return value:** an **iterator** that points to the first of the newly inserted elements.



// erasing from vector

|  |
| --- |
| myvector contains: 4 5 6 8 9 10 |

#include <iostream> #include <vector> int main() { std::vector<int> myvector; // set some values (from 1 to 10)

for (int i = 1; i <= 10; i++) myvector.push\_back(i);

// erase the 7th element myvector.erase(myvector.begin() + 6);

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

std::cout << "myvector contains:"; for (unsigned i = 0; i<myvector.size(); ++i)

std::cout << ' ' << myvector[i];

std::cout << '\n'; return 0;

}

|  |
| --- |
| myvector contains: 501 502 503 300 300 400 400 200 100 100 100 |

#include <iostream> #include <vector> int main(){ std::vector<int> myvector(3, 100); std::vector<int>::iterator it; it = myvector.begin(); it = myvector.insert(it, 200); myvector.insert(it, 2, 300);

// "it" no longer valid, get a new one: it = myvector.begin();

std::vector<int> anothervector(2, 400);

myvector.insert(it + 2, anothervector.begin(), anothervector.end());

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

std::cout << "myvector contains:"; for (it = myvector.begin(); it<myvector.end(); it++) std::cout << ' ' << \*it; return 0;}

|  |
| --- |
| myvector contains: 1 2 3 4 5 100 100 100 0 0 0 0 |

#include <iostream> #include <vector> int main() { std::vector<int> myvector; // set some initial content: for (int i = 1; i<10; i++) myvector.push\_back(i); myvector.resize(5); myvector.resize(8, 100); myvector.resize(12);

std::cout << "myvector contains:"; for (int i = 0; i<myvector.size(); i++) std::cout << ' ' << myvector[i]; std::cout << '\n'; return 0;}

# The sort Algorithm

* Sorts the elements in the range [first,last) into ascending order.
* void sort (Iterator first, Iterator last);
* #include <algorithm>

#include<iostream>

|  |
| --- |
| Write a program that can read any number of integers from the user, stores them in a vector, sorts them, and print the result. |

#include<vector>

#include<algorithm> using namespace std; int main() { int input; vector<int> ivec;

// input while (cin >> input ) ivec.push\_back(input); **sort(ivec.begin(), ivec.end());** vector<int>::iterator it;

for ( it = ivec.begin(); it != ivec.end(); ++it )

cout << \*it << " ";

return 0;

}

# Generate random number

## • **int rand (void);**

* Returns a pseudo-random integral number in the range between 0 and RAND\_MAX, which is a constant defined in <cstdlib>.
* This number is generated by an algorithm that returns a sequence of apparently non-related numbers each time it is called.
* This algorithm uses a **seed** to generate the series, which should be initialized to some distinctive value using function **srand**.
* Notice though that this modulo operation does not generate **uniformly distributed random numbers** in the span
* A typical way to generate trivial pseudo-random numbers in a determined range using rand is to use the modulo of the returned value by the range span and add the initial value of the range:
  + v1 = rand() % 100; // v1 in the range 0 to 99
  + v2 = rand() % 100 + 1; // v2 in the range 1 to 100
* void **srand** (unsigned int seed);
  + **Initialize** random number generator
  + The pseudo-random number generator is initialized using the argument passed as seed.
  + For every different seed value used in a call to srand, the pseudorandom number generator can be expected to **generate a different succession of results in the subsequent calls to rand**.
  + Two different initializations with the same seed will generate the same succession of results in subsequent calls to rand.
  + If seed is set to **1**, the generator is reinitialized to its **initial value** and produces the same values as before any call to rand or srand.
  + In order to generate random-like numbers, srand is usually initialized to some distinctive runtime value, like the value returned by function **time** (declared in header <ctime>). This is distinctive enough for most trivial randomization needs.

#include <iostream>

|  |  |
| --- | --- |
| #include <cstdlib> | /\* srand, rand \*/ |
| #include <ctime> | /\* time \*/ |

using namespace std; int main() {

cout << "First number: " << rand() << endl;

srand(time(NULL)); for (int i = 0; i <5; i++) cout << "Random number: " << rand() << endl;

srand(1); cout << "Again the first number: " << rand(); getchar(); return 0;

}

# Passing arguments by reference

* When passing arguments by value, the only way to return a value back to the caller is via the function’s **return** value.
* One way to allow functions to modify the value of argument is by using **pass by reference.**

void AddOne(int &y) // y is a reference variable {y = y + 1;}

* When the function is called, y will become a reference to the argument. **Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the argument.**
* [More: http://www.learncpp.com/cpp-tutorial/73-passingarguments-by-reference/](http://www.learncpp.com/cpp-tutorial/73-passing-arguments-by-reference/)

|  |
| --- |
| X = 5  X = 7 |

#include<iostream> using namespace std;

void passByReference(int &y) // y is a reference

{ y = 7;}

void passByValue(int y) // y is a copy

{ y = 6;}

int main()

{ int x = 5; passByValue(x); cout << "x = " << x << endl; passByReference(x); cout << "x = " << x << endl; getchar(); return 0;

}

|  |
| --- |
| When a vector is passed as a parameter to some function, a copy of vector is actually created. |

#include <iostream>

#include <vector> #include <algorithm> using namespace std;

|  |
| --- |
| Output: 5 6 7  3 6 7 |

void copy\_vector(vector<int> v2) { v2.at(0) = 2;}

void pass\_vector(vector<int> &v3)

{ v3.at(0) = 3;}

int main() { vector<int> v;

v.push\_back(5);v.push\_back(6); v.push\_back(7); vector<int>::iterator it;

copy\_vector(v); for (it = v.begin(); it != v.end(); )

cout << \*it++ << " ";

cout << endl; pass\_vector(v); for (it = v.begin(); it != v.end(); ) cout << \*it++ << " "; return 0;}

# setw

* **setw (int n);**
* Set field width
* Sets the field width to be used on output operations.

|  |  |
| --- | --- |
| #include <iostream> | // std::cout, std::endl |
| #include <iomanip> | // std::setw |

int main() {

std::cout << std::setw(4); std::cout << 55; return 0;

}

Write a C++ program to enter 10 random numbers between 5 and 9 into a vector. Then call a function removeEven(vector<int>& v) to remove all even numbers. Finally print the vector

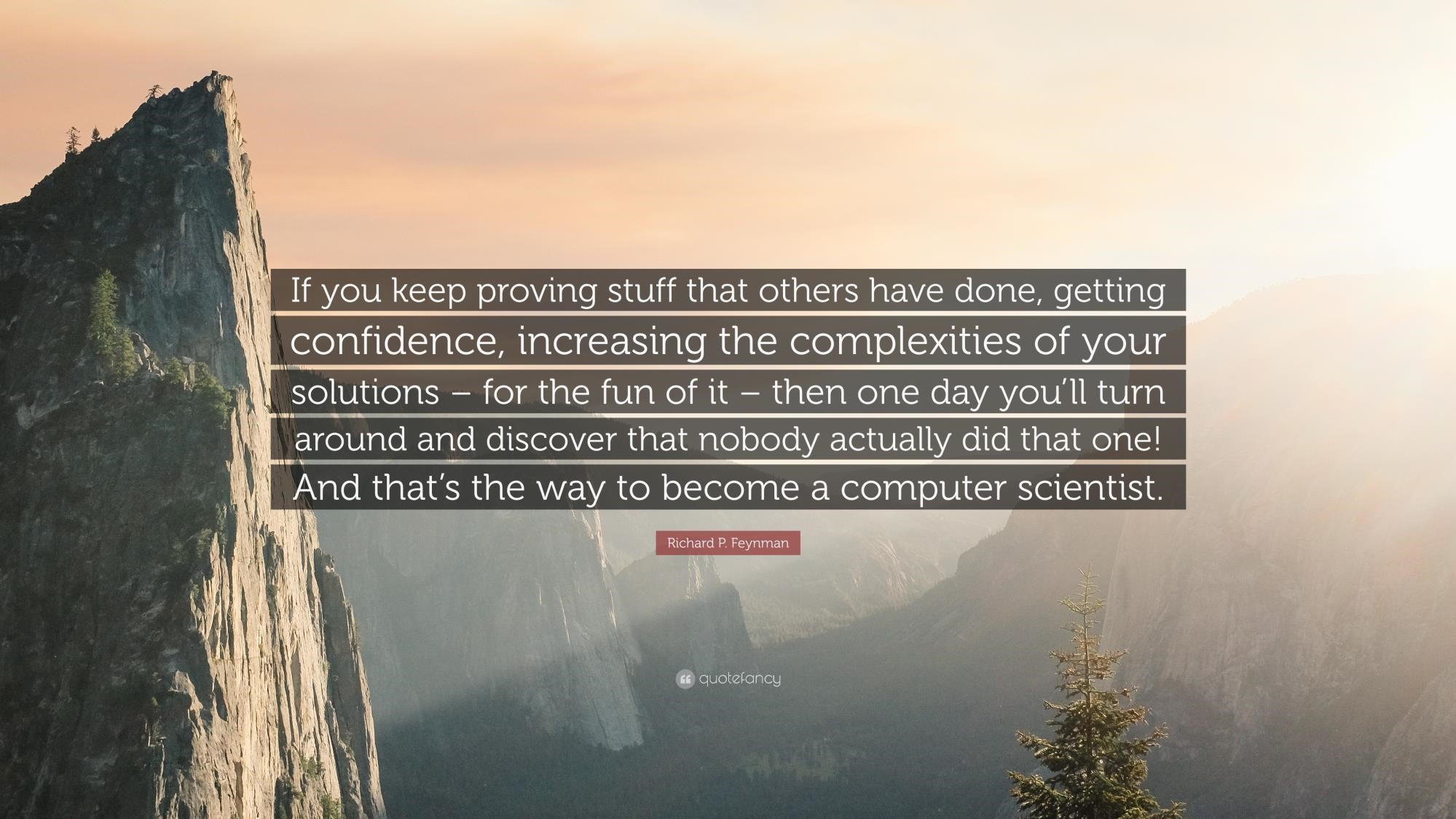
#include <iostream>

#include <vector> #include <cstdlib> using namespace std; void removeEven(vector < int > & v2) { vector < int > ::iterator it; for (it = v2.begin(); it != v2.end();) if ( \* it % 2 == 0) it = v2.erase(it); else it++;}

int main() {

int random; vector < int > v1; vector < int > ::iterator it; srand(time(NULL)); for (int i = 0; i < 10; i++) { random = 5 + rand() % 5; cout << random << " "; v1.push\_back(random);}

removeEven(v1); cout << "\n After removing even numbers:"; for (it = v1.begin(); it != v1.end(); it++) cout << \* it << " "; return 0;}

If you keep proving stuff that others have done, getting confidence, increasing the complexities of your solutions — for the fun of it — then one day you'll turn around and discover that nobody actually did that one! And that's the way to become a computer scientist.

*Richard Feynman, Lectures on Computation*

# Searching and Sorting Algorithms

* InputIterator **find** (InputIterator first, InputIterator last, const T& val);
  + Returns an iterator to the **first element** in the range [first,last) that compares equal to val. If no such element is found, the function returns **last**.
* InputIterator **find\_if** (InputIterator first, InputIterator last, UnaryPredicate pred);
  + Returns an iterator to the first element in the range [first,last) for which pred returns true. If no such element is found, the function returns last.
  + **pred**: Unary function that accepts an element in the range as argument and returns a value convertible to **bool**. The value returned indicates whether the element is considered a match in the context of this function.
* bool **binary\_search** (ForwardIterator first, ForwardIterator last, const T& val);
  + Returns true if any element in the range [first,last) is equivalent to val, and false otherwise.
  + The elements in the range shall already be **sorted**.
* void **sort** (RandomAccessIterator first, RandomAccessIterator last);
  + Sorts the elements in the range [first,last) into ascending order.

#include <iostream> #include <set> using namespace std; int main() { set<int> s; set<int>::iterator it;

for (int i = 1; i <= 9; i++)

s.insert(i);

s.erase(5);

it = s.begin();

++it;

s.erase(it, s.find(7));

**Output: 1 7 8 9**

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

return 0; }

#include <iostream>

|  |
| --- |
| Output: 0 1 2 8 3 4 5 1 |

#include <list> #include <algorithm> bool IsOdd(int i) {return ((i % 2) == 1);} using namespace std; int main() { list<int> li; for (int nCount = 0; nCount < 6; nCount++) li.push\_back(nCount);

list<int>::const\_iterator it; it = **find**(li.begin(), li.end(), 3); li.insert(it, 8);

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

cout<< \*(**find\_if**(li.begin(), li.end(), IsOdd)); return 0;

}

#include <iostream>

|  |
| --- |
| -5 -3 0 2 6 7 |

#include <vector> #include <algorithm> int main() { using namespace std; vector<int> vect; vect.push\_back(7); vect.push\_back(-3); vect.push\_back(6); vect.push\_back(2); vect.push\_back(-5); vect.push\_back(0); **sort(vect.begin(), vect.end());**

vector<int>::const\_iterator it; for (it = vect.begin(); it != vect.end(); it++) cout << \*it << " ";

cout << endl;

return 0;

}

#include <iostream>

#include <algorithm>

|  |
| --- |
| Found 16 at location 4  The first value greater than 10 is 17  12 was not found in v 12 was found in v |

#include <vector> #include <iterator> using namespace std; bool greater10(int value) {return value > 10;} int main() { const int SIZE = 10;

int a[SIZE] = { 10, 2, 17, 5, 16, 8, 12, 11, 20, 7 }; vector<int> v(a, a + SIZE); // copy of a vector<int>::iterator location;

location = **find**(v.begin(), v.end(), 16); if (location != v.end()) cout << "Found 16 at location " << (location - v.begin()) << endl;

else cout << "16 not found \n";

location = **find\_if**(v.begin(), v.end(), greater10); if (location != v.end()) cout << "The first value greater than 10 is " << \*location << endl;

else cout << "No values greater than 10 were found \n";

if (**binary\_search**(v.begin(), v.end(), 12))

cout << "12 was found in v \n";

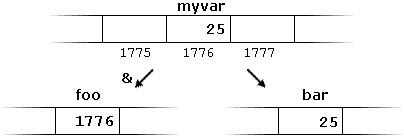
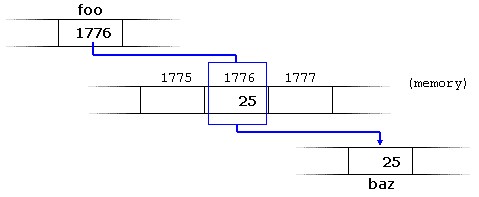
else

cout << "12 was not found in v \n";

**sort**(v.begin(), v.end()); if (binary\_search(v.begin(), v.end(), 12)) cout << "12 was found in v \n";

else cout << "12 was not found in v \n"; return 0;}

# Pointers

* The declaration of pointers follows this syntax:
  + type \* name;
  + int \*foo; //declaring a pointer
* The variable that stores the address of another variable (like foo in the previous example) is what in C++ is called a **pointer**.
* The address of a variable can be obtained by preceding the name of a variable with an ampersand sign **(&),** known as address-of operator. For example:
  + foo = &myvar;
* This would assign the address of variable myvar to foo; by preceding the name of the variable myvar with the address-of operator (&), we are no longer assigning the content of the variable itself to foo, but its address.
* More details: <http://www.cplusplus.com/doc/tutorial/pointers/>
* Assume myvar is placed during runtime in the memory address 1776.
  + myvar = 25;
  + foo = &myvar;
  + bar = myvar;
* Pointers can be used to access the variable they point to directly. This is done by preceding the pointer name with the **dereference operator (\*)**.
  + int baz = \*foo; 
* Thus, & and \* have sort of **opposite** meanings: An address obtained with & can be dereferenced with \*.

|  |
| --- |
| firstvalue is 10 secondvalue is 20 |

#include <iostream> using namespace std;

int main() {

int firstvalue, secondvalue; int \* mypointer;

mypointer = &firstvalue; \*mypointer = 10; mypointer = &secondvalue; \*mypointer = 20;

cout << "firstvalue is " << firstvalue << '\n'; cout << "secondvalue is " << secondvalue << '\n'; return 0;

}

|  |
| --- |
| firstvalue is 10 secondvalue is 20 |

#include <iostream> using namespace std;

int main()

{

int firstvalue = 5, secondvalue = 15; int \* p1, \*p2;

p1 = &firstvalue; // p1 = address of firstvalue p2 = &secondvalue; // p2 = address of secondvalue \*p1 = 10; // value pointed to by p1 = 10

\*p2 = \*p1; // value pointed to by p2 = value pointed to by p1 p1 = p2; // p1 = p2 (value of pointer is copied)

\*p1 = 20; // value pointed to by p1 = 20

cout << "firstvalue is " << firstvalue << '\n'; cout << "secondvalue is " << secondvalue << '\n'; return 0;

}

# Pointers and arrays

* The concept of arrays is related to that of pointers. In fact**, arrays work very much like pointers** to their first elements, and, actually, an array can always be implicitly converted to the pointer of the proper type. For example, consider these two declarations:
  + int myarray [20];
  + int \* mypointer;
* The following assignment operation would be valid:
  + mypointer = myarray;
* After that, mypointer and myarray would be equivalent and would have very similar properties. The main difference being that mypointer can be assigned a different address, whereas myarray can never be assigned anything, and will always represent the same block of 20 elements of type int. Therefore, the following assignment would **not** be valid:
  + myarray = mypointer;

|  |
| --- |
| 10, 20, 30, 40, 50, |

#include <iostream> using namespace std;

int main() { int numbers[5]; int \* p; p = numbers; \*p = 10; p++; \*p = 20; p = &numbers[2]; \*p = 30; p = numbers + 3; \*p = 40; p = numbers; \*(p + 4) = 50; for (int n = 0; n<5; n++) cout << numbers[n] << ", ";

return 0; }

# Pointers to functions

• C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function. Pointers to functions are declared with the same syntax as a regular function declaration, except that the **name of the function is enclosed between parentheses () and an asterisk (\*) is inserted before the name:**

# Pointer to function – example 1

|  |
| --- |
| 15  20 |

#include <iostream> using namespace std;

void one(int a, int b) { cout << a + b << "\n"; } void two(int a, int b) { cout << a\*b << "\n"; }

int main()

{ void(\*fptr)(int, int); // a function pointer to voids with two int params

fptr = one; //fptr -> one fptr(12, 3); //=> one(12, 3)

fptr = two; //fptr -> two fptr(5, 4); //=> two(5, 3) return 0;}

# Pointer to function – example 2

#include <iostream> using namespace std;

Output: a = 12 and b = 8

int add(int first, int second) {return first + second;}

int subtract(int first, int second) {return first - second;}

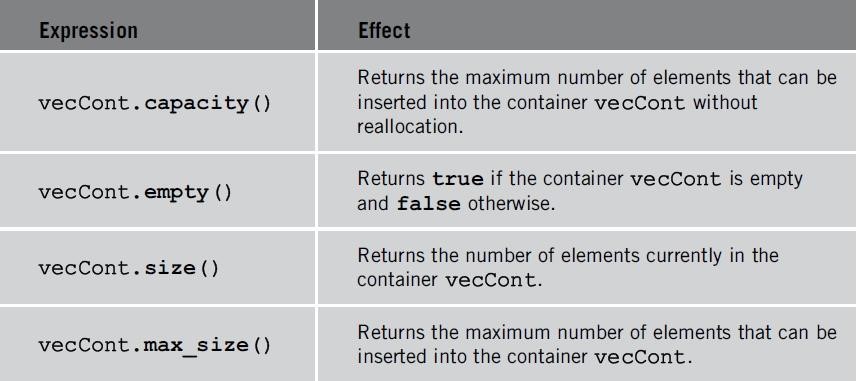
int operation(int first, int second, int(\*functocall)(int, int)) {return functocall(first, second);}

int main()

{ int a, b;

a = operation(7, 5, add); b = operation(20, a, subtract); cout << "a = " << a << " and b = " << b << endl; return 0;}

## Functions to determine the **size** of a vector container



// comparing size, capacity and max\_size

|  |
| --- |
| size: 100 capacity: 141 max\_size: 1073741823 |

#include <iostream> #include <vector>

int main() { std::vector<int> myvector;

// set some content in the vector: for (int i = 0; i<100; i++) myvector.push\_back(i);

std::cout << "size: " << myvector.size() << '\n'; std::cout << "capacity: " << myvector.capacity() << '\n'; std::cout << "max\_size: " << myvector.max\_size() << '\n'; return 0;

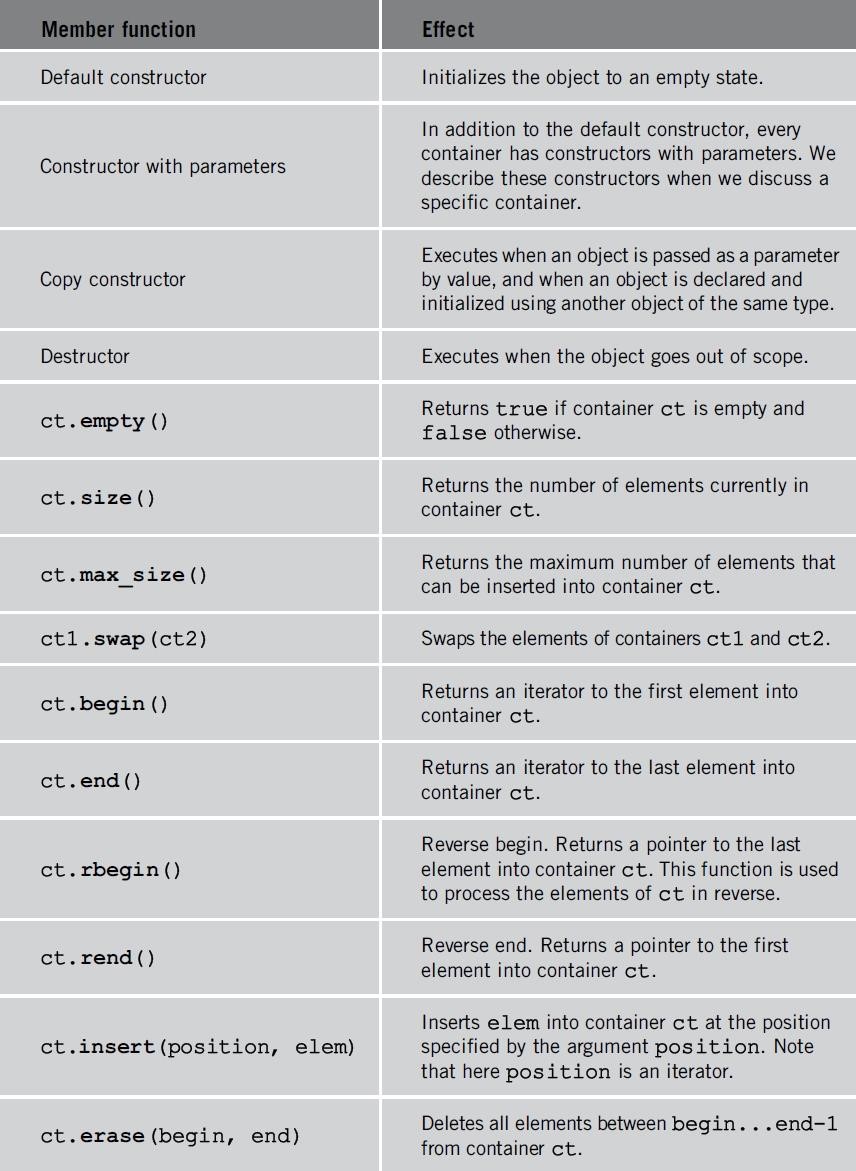
}

Member Functions Common to **All** Containers

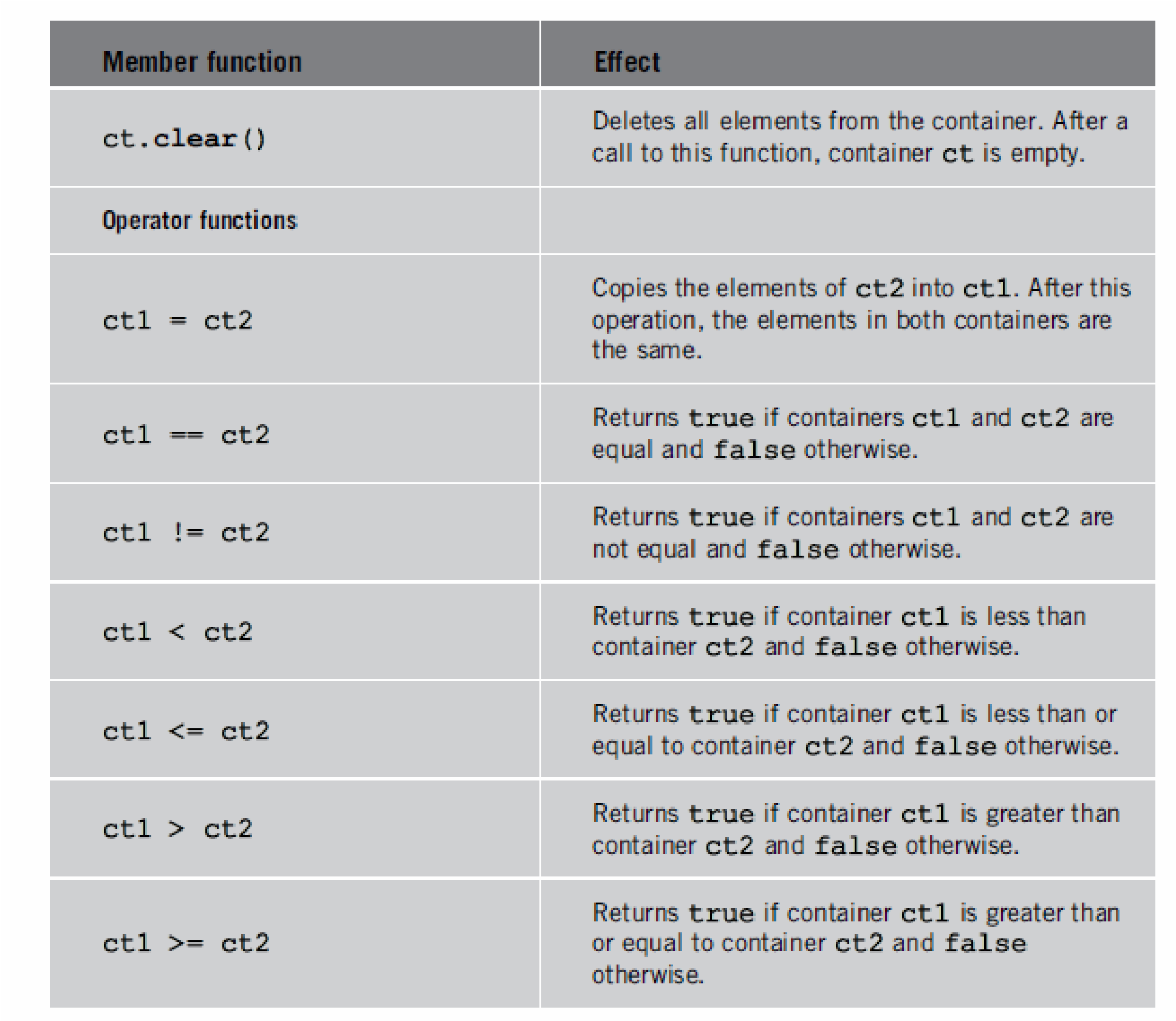
* Examples
  + Default constructor
  + Several constructors with parameters
  + Destructor
  + Function inserting an element into a container
* Class encapsulates data, operations on that data
  + Into a single unit
* Every container is a **class**
  + Several operations directly defined for a container
  + Provided as part of class definition

Member functions common to **all** containers

the element past the end

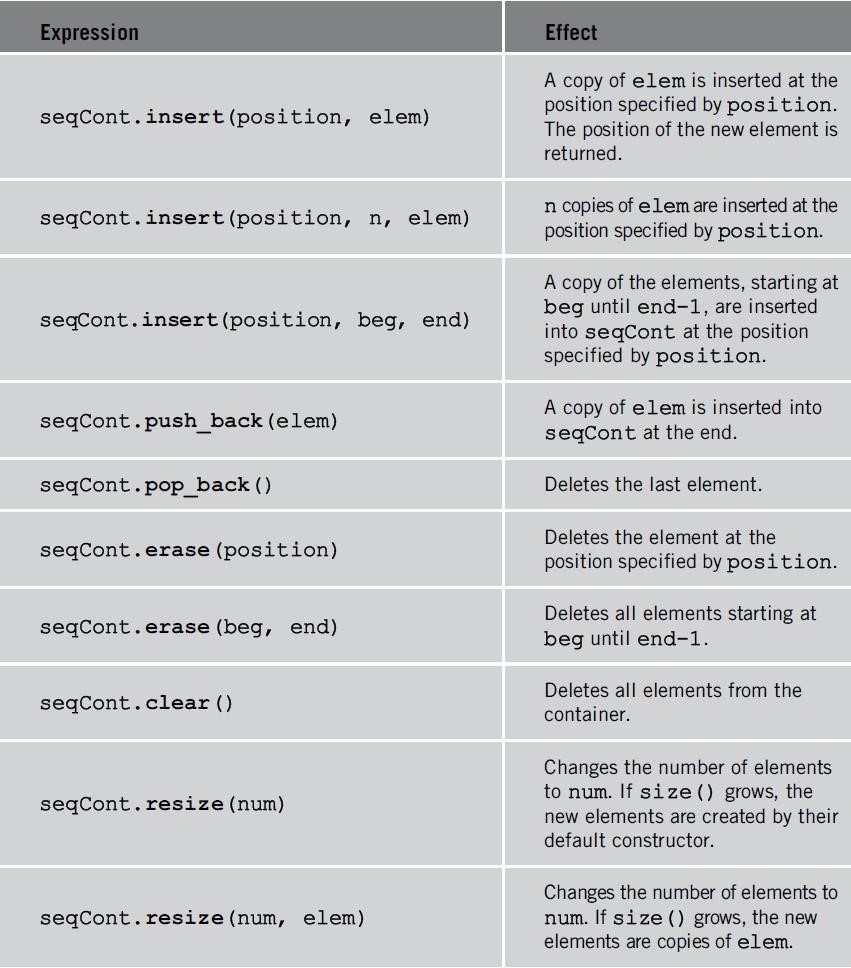


Member functions common to **all** containers

Compares the content

<http://en.cppreference.com/w/cpp/container/vector/operator_cmp>

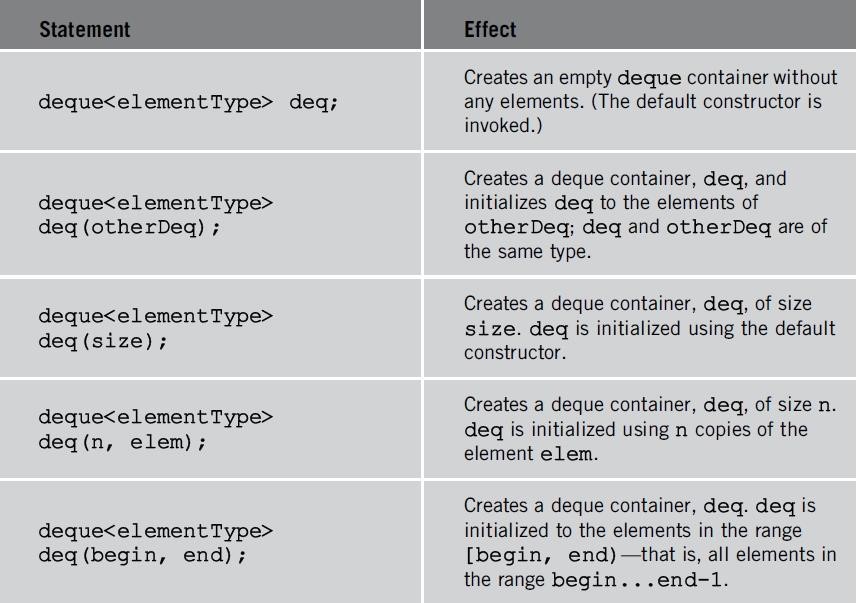
## Member Functions Common to Sequence Containers



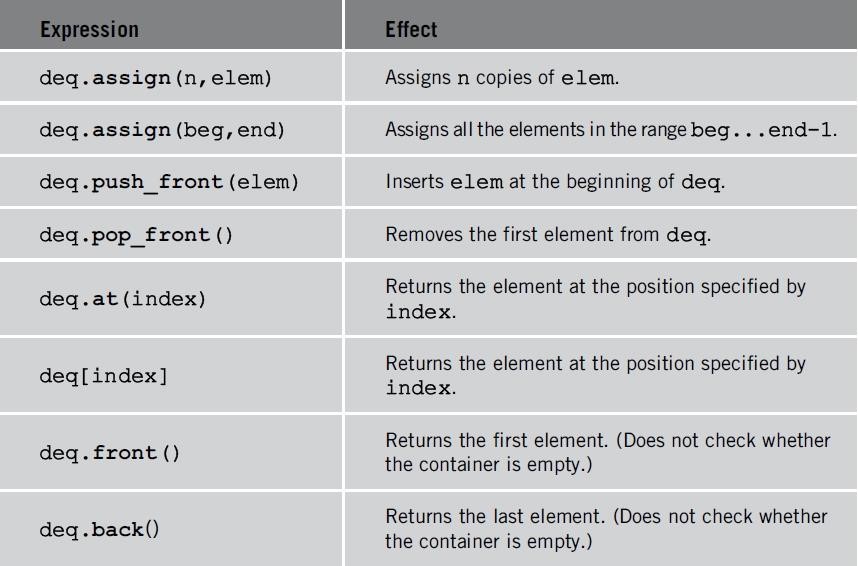
# Sequence Container: deque

* **Deque: double-ended queue**
* Implemented as **dynamic arrays**
* Can expand in **either direction**
* Therefore, they provide a functionality similar to **vectors**, but with efficient insertion and deletion of elements also at the beginning of the sequence, and not only at its end.
* Both vectors and deques provide a very similar interface and can be used for similar purposes, but **internally** both work in quite different ways: While vectors use a single array that needs to be occasionally reallocated for growth, the elements of a deque can be scattered in different chunks of storage, with the container keeping the necessary information internally to provide direct access to any of its elements in constant time and with a uniform sequential interface (through iterators).
* For operations that involve frequent insertion or removals of elements at positions other than the beginning or the end, **deques perform worse** and have less consistent iterators and references than lists.
* Header file deque contains
  + Definition of the class deque
  + Functions to implement various operations on a deque object

## Various ways to declare a deque object



Various operations that can be performed on a deque object



// vector assign

|  |
| --- |
| Size of first: 7  Size of second: 5  Size of third: 3 |

#include <iostream> #include <vector> int main() {

std::vector<int> first; std::vector<int> second; std::vector<int> third; first.assign(7, 100); // 7 ints with a value of 100

std::vector<int>::iterator it; it = first.begin() + 1; second.assign(it, first.end() - 1); // the 5 central values of first

int myints[] = { 1776,7,4 }; third.assign(myints, myints + 3); // assigning from array.

std::cout << "Size of first: " << int(first.size()) << '\n'; std::cout << "Size of second: " << int(second.size()) << '\n'; std::cout << "Size of third: " << int(third.size()) << '\n'; return 0;

}

// deque::push\_front

#include <iostream>

#include <deque>

mydeque contains: 300 200 100 100

int main()

{

std::deque<int> mydeque(2, 100); mydeque.push\_front(200); mydeque.push\_front(300);

std::cout << "mydeque contains:"; for (std::deque<int>::iterator it = mydeque.begin(); it

!= mydeque.end(); ++it) std::cout << ' ' << \*it; std::cout << '\n';

return 0;

}

# Sequence Container: list

* Lists are sequence containers that allow constant time insert and erase operations anywhere within the sequence, and **iteration in both directions**.
* A list is a special type of sequence container called a **doubly linked list** where each element in the container contains pointers that point at the next and previous elements in the list.
* Lists only provide access to the start and end of the list -**there is no random access provided.**

# Iterating through a vector

|  |
| --- |
| Output: 0 1 2 3 4 5 |

#include <iostream> #include <vector> int main() { using namespace std; vector<int> vect; for (int nCount = 0; nCount < 6; nCount++) vect.push\_back(nCount);

vector<int>::const\_iterator it; it = vect.begin(); while (it != vect.end()) cout << \*it++ << " ";

return 0;

}

# Iterating through a deque

|  |
| --- |
| Output: 8 9 10 0 1 2 |

#include <iostream> #include <deque> int main() { using namespace std;

deque<int> deq; for (int nCount = 0; nCount < 3; nCount++)

{ deq.push\_back(nCount); deq.push\_front(10 - nCount);

}

for (int nIndex = 0; nIndex < deq.size(); nIndex++) cout << deq[nIndex] << " ";

return 0;

}

# Iterating through a list

#include <iostream> #include <list> int main()

{ Output: 0 1 2 3 4 5 using namespace std; list<int> li; for (int nCount = 0; nCount < 6; nCount++) li.push\_back(nCount);

|  |
| --- |
| Note the code is almost identical to the vector case, even though vectors and lists have almost completely different internal implementations! |

list<int>::const\_iterator it; it = li.begin(); while (it != li.end()) cout << \*it++ << " "; return 0;}

// inserting into a list

#include <iostream>

|  |
| --- |
| mylist contains: 1 10 20 30 30 20 2 3 4 5 |

#include <list> #include <vector> int main() {

std::list<int> mylist; std::list<int>::iterator it; for (int i = 1; i <= 5; ++i) mylist.push\_back(i);

it = mylist.begin();

++it; mylist.insert(it, 10); mylist.insert(it, 2, 20);

--it; std::vector<int> myvector(2, 30);

mylist.insert(it, myvector.begin(), myvector.end()); std::cout << "mylist contains:";

for (it = mylist.begin(); it != mylist.end(); ++it) std::cout << ' ' << \*it; std::cout << '\n'; return 0;}

# list operations

* **void remove (const value\_type& val);**
  + remove all elements with specific value
* **Sort**
  + Sorts the elements in the list, altering their position within the container.
  + **void sort();**
* Sorts the elements in the list, altering their position within the container.
  + **void sort (Compare comp);**

#include <iostream>

#include <list>

#include <string> #include <cctype> bool compare\_nocase(const std::string& first, const std::string& second) { unsigned int i = 0; while ((i<first.length()) && (i<second.length()))

{ if (tolower(first[i])<tolower(second[i])) return true; else if (tolower(first[i])>tolower(second[i])) return false; ++i; }

return (first.length() < second.length());

}

|  |
| --- |
| mylist contains: Three one two mylist contains: one Three two |

int main()

{ std::list<std::string> mylist; std::list<std::string>::iterator it; mylist.push\_back("one"); mylist.push\_back("two"); mylist.push\_back("Three"); mylist.sort();

std::cout << "mylist contains:";

for (it = mylist.begin(); it != mylist.end(); ++it) std::cout <<

' ' << \*it; std::cout << '\n'; mylist.sort(compare\_nocase);

std::cout << "mylist contains:"; for (it = mylist.begin(); it != mylist.end(); ++it) std::cout <<

' ' << \*it;

std::cout << '\n'; return 0; }

# list operations - Merge

* **void merge (list& x);**
  + Merges x into the list by transferring all of its elements at their respective ordered positions into the container.
  + This effectively **removes all** the elements in x (which becomes **empty**), and inserts them into their ordered position within container (which expands in size by the number of elements transferred).
  + This function requires that the list containers have their elements already **ordered** by value (or by comp) before the call.
* **void merge (list& x, Compare comp);**
  + Have the same behavior, but take a specific predicate (comp) to perform the comparison operation between elements.

#include <iostream> #include <list> bool mycomparison(double first, double second)

{return ((first)<(second));} int main()

{

|  |
| --- |
| first contains: 1.4, 2.2, 2.9, 3.1, 3.7, 7.1, first contains: 1.4, 2.1, 2.2, 2.9, 3.1, 3.7, 7.1, |

std::list<double> first, second; first.push\_back(3.1); first.push\_back(2.2); first.push\_back(2.9); second.push\_back(3.7); second.push\_back(7.1); second.push\_back(1.4); first.sort(); second.sort(); first.merge(second);

std::cout << "first contains: "; for (std::list<double>::iterator it = first.begin(); it != first.end(); ++it) std::cout << \*it << ", "; std::cout << '\n';

second.push\_back(2.1);

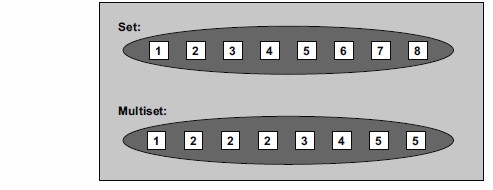
first.merge(second, mycomparison); std::cout << "Now first contains: "; for (std::list<double>::iterator it = first.begin(); it != first.end(); ++it) std::cout << \*it << ", "; return 0;}

# Associative Containers

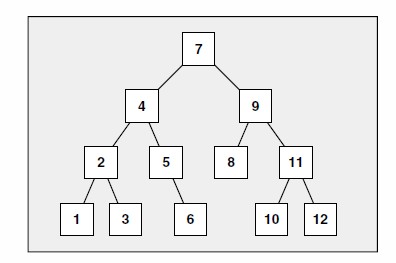
* Associative contains are containers that **automatically sort** their inputs when those inputs are inserted into the container. By default, associative containers compare elements using operator< (less than).
* Elements in associative containers are referenced by their **key** and not by their absolute position in the container.
* A **set** is a container that stores **unique** elements.
* A **multiset** is a set where **duplicate** elements are allowed.
* A **map** (also called an associative array) is a set where each element is a pair, called a **key/value** pair. The key is used for sorting and indexing the data, and must be unique. The value is the actual data.
* A **multimap** (also called a dictionary) is a map that allows **duplicate** keys. Real-life dictionaries are multimaps: the key is the word, and the value is the meaning of the word.

# Sets

* Sets are containers that store **unique** elements following a specific **order**.
* Multisets allow duplicates.
* To use a set or a multiset, you must include the header file **<set>.**



* Sets are typically implemented as **binary search trees.**

Internal Structure of Sets and Multisets

* The value of the elements in a set cannot be modified once in the container (the elements are always **const**), because doing so might compromise the correct **order,** but they can be inserted or removed from the container.
* As with all associative container classes, the iterators are **bidirectional iterators.**
* If you want to use a **sort** criterion other than the default, you must specify this option when the container is declared.
* Include header file <**functional>**
* set<int> intSet; //ascending order
* set<int, **greater**<int> > otherIntSet; //descending order
* multiset<string> stringMultiSet;
* multiset<string, **less**<string> > otherStringMultiSet;

# Operations to insert elements in a set

**(1) mySet.insert(val)**

– Inserts a copy of val into mySet.

## **(2) mySet.insert(iteratorPos, val)**

* Inserts a copy of val into mySet.
* The position where val is inserted is returned.
* The first parameter, **hints** at where to begin the search for insert.

## **(3) mySet.insert(iteratorBegin, iteratorEnd);**

– Inserts a copy of all the elements into mySet starting at the position iteratorBegin until iteratorEnd-1.

#include <iostream>

|  |
| --- |
| **Output: 7 8 9 90 80 70** |

#include <set> #include <functional> using namespace std; int main() {

set<int, less<int>> s1; //ascending order set<int, greater<int>> s2; //descending order set<int>::iterator it; for (int i = 7; i <= 9; i++)

{s1.insert(i); s2.insert(i \* 10);}

for (it = s1.begin(); it != s1.end(); ++it) cout << \*it << " "; cout << endl;

for (it = s2.begin(); it != s2.end(); ++it) cout << \*it << " "; return 0;}

#include <iostream>

|  |
| --- |
| **Output: 2 7 8 9 10** |

#include <set> #include <vector> using namespace std; int main() { vector<int> v; set<int> s; set<int>::iterator it;

v.push\_back(2);

v.push\_back(10);

for (int i = 7; i <= 9; i++) s.insert(i);

s.insert(v.begin(), v.end());

for (set<int>::iterator it = s.begin(); it != s.end(); ++it) cout << \*it << " "; return 0;}

# Operations to remove elements from a set

## **(1) mySet.erase(val);**

* Deletes all the elements with the value val.
* The number of deleted elements is returned.

## **(2) mySet.erase(iteratorPos);**

* Deletes the element at the position specified by the iterator position.
* **Return an iterator to the element that follows the last element removed (or set::end, if the last element was removed).**

## **(3) mySet.erase(iteratorBegin, iteratorEnd);**

– Deletes all the elements starting at the position iteratorBegin **until iteratorEnd-1.**

## **(4) mySet.clear();**

* Deletes all the elements from mySet.
* After this operation, mySet is empty.
* **mySet.size()** 
  + Returns the current number of elements
* **mySet.count(val)**
  + Returns the number of elements with value val

# Iterating through a set

|  |
| --- |
| Output: -6 -4 1 2 7 8 |

#include <iostream> #include <set> int main() { using namespace std; set<int> myset; myset.insert(7); myset.insert(2); myset.insert(-6); myset.insert(8); myset.insert(1); myset.insert(-4);

set<int>::const\_iterator it; it = myset.begin(); while (it != myset.end()) cout << \*it++ << " ";

cout << endl; return 0;

}

# Iterating through a map

#include <iostream>

|  |
| --- |
| Output: 1=orange 2=peach 3=grapes 4=apple |

#include <map> #include <string> int main() {

using namespace std; map<int, string> mymap; mymap.insert(make\_pair(4, "apple")); mymap.insert(make\_pair(1, "orange")); mymap.insert(make\_pair(3, "grapes")); mymap.insert(make\_pair(2, "peach"));

map<int, string>::const\_iterator it; it = mymap.begin(); while (it != mymap.end())

{cout << it->first << "=" << it->second << " "; it++;}

return 0;}

## Sieve of Eratosthenes (Siv of air-uh Taws-thuh neeze)

• A simple algorithm to find **prime numbers** from 2 to N • Examples

* <http://www.algolist.net/Algorithms/Number_theoretic/Sieve_of_Eratosthenes>
* <http://www.visnos.com/demos/sieve-of-eratosthenes>
* <https://www.youtube.com/watch?v=V08g_lkKj6Q>
* <https://www.youtube.com/watch?v=9m2cdWorIq8>

# Iterators

* An iterator is any object that points to some element in a range of elements (such as an array or a container), has the ability to iterate through the elements of that range using a set of operators (with at least the increment (++) and dereference (\*) operators). • It represents a **position** in the container
* A **pointer** is a form of iterator.
* Iterators must be implemented on a per-class basis, because the iterator does need to know how a class is implemented. Thus iterators are always tied to **specific container** classes.
  + Each STL container defines what iterators it can return.
* Instead of operating on specific data types, **algorithms** are defined to operate on a range specified by a type of iterator.
  + Each algorithm specifies what class of iterators it requires.
* Iterators can be **generated** by STL container member functions, such as begin() and end().
* All STL containers (but not the adapters) provide at least two types of iterators:

## **(1) iterator**

– Every container (sequence or associative) contains an iterator – **Read/Write** iterator

## **(2) const\_iterator**

* Prevents iterator from modifying elements of container declared as constant
* Every container contains const\_iterator – **Read-only** iterator

# Iterator categories

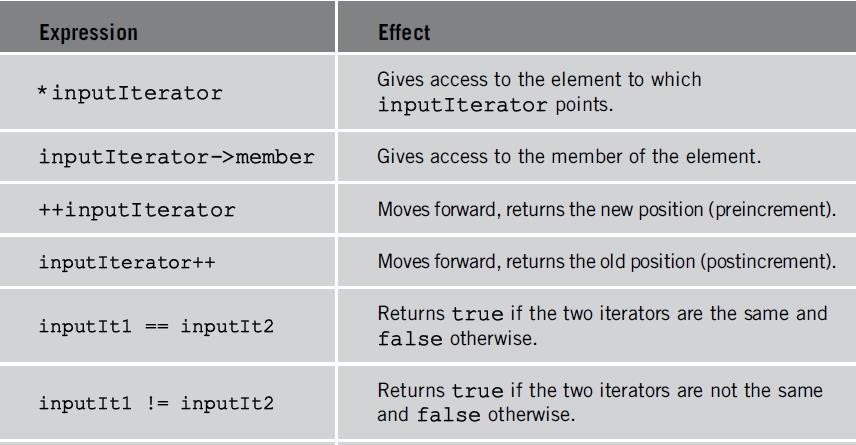
1. Input iterators
2. Output iterators
3. Forward iterators
4. Bidirectional iterators
5. Random access iterators

# (1) Input Iterators

* **Read** data from an input stream
* Step **forward** element-by-element
* Return values element-by-element
* Input iterators can read elements **only once**.
* Example

–InputIterator **find** (**InputIterator** first, **InputIterator** last, const T& val);

## **Operations on an input iterator**

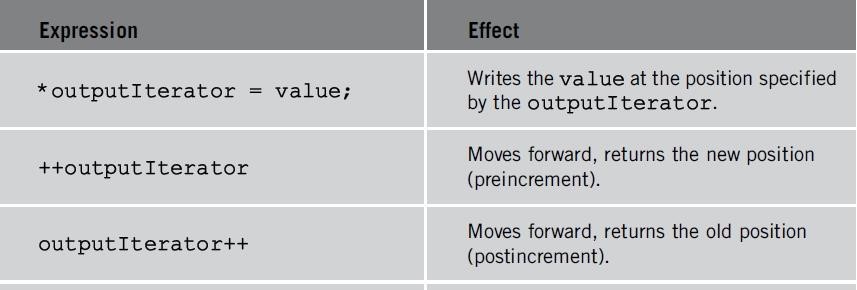


# (2) Output Iterators

* **Write** data to an output stream
* Step forward element-by-element
* As with input iterators, you can't use an output iterator to iterate twice over the same range.
* Example

–OutputIterator **copy** (InputIterator first, InputIterator last, **OutputIterator** result);

# Operations on an output iterator



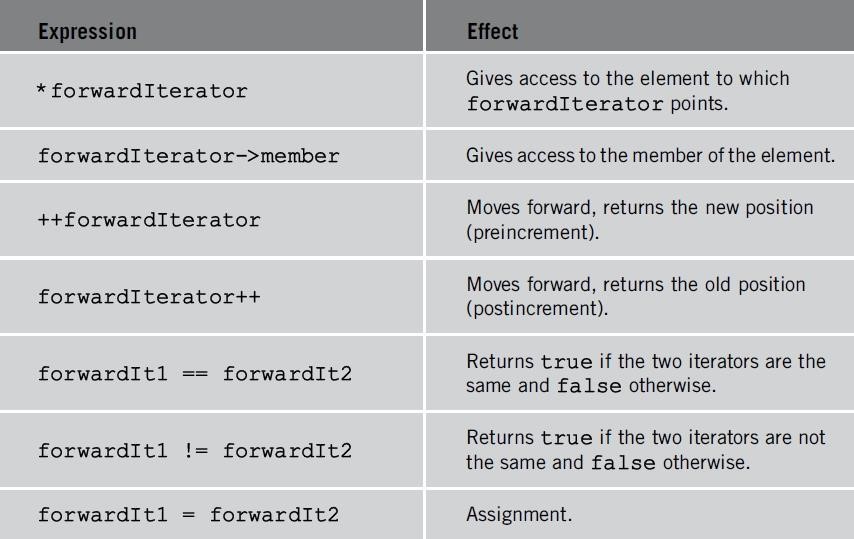


# (3) Forward Iterators

* Combines the functionality of the **input** and **output** iterators.
* Unlike input and output iterators, forward iterators can refer to the same element in the same collection and process the same element more than once (can be used in **multipass** algorithms).
* Example

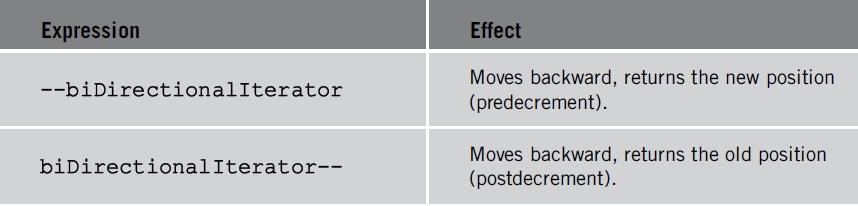
–bool **binary\_search** (**ForwardIterator** first, ForwardIterator last, const T& val);

# Operations on a forward iterator



# (4) Bidirectional Iterators

* Allow algorithms to pass through the elements **forward** and **backward**.
* Operations defined for **forward iterators** applicable to bidirectional Iterators
* This type of iterator can used with the sequence and associative containers.
* Additional operations on a bidirectional iterator



# (5) Random Access Iterators

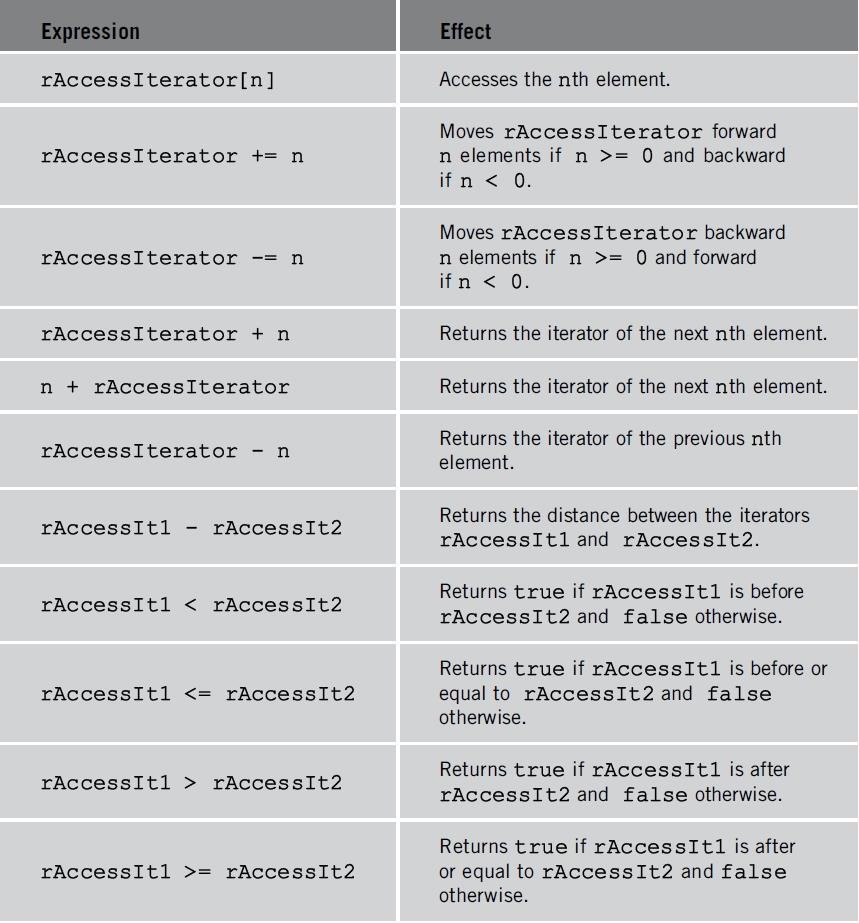
* Bidirectional iterators that can **randomly** process container elements
* Can be used with containers of type:

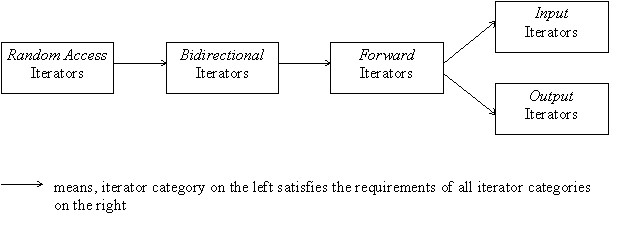
– vector , deque , string, and arrays

* Operations defined for bidirectional iterators applicable to random access iterators

–void **sort** (**RandomAccessIterator** first, RandomAccessIterator last);

## Additional operations on a random access iterator

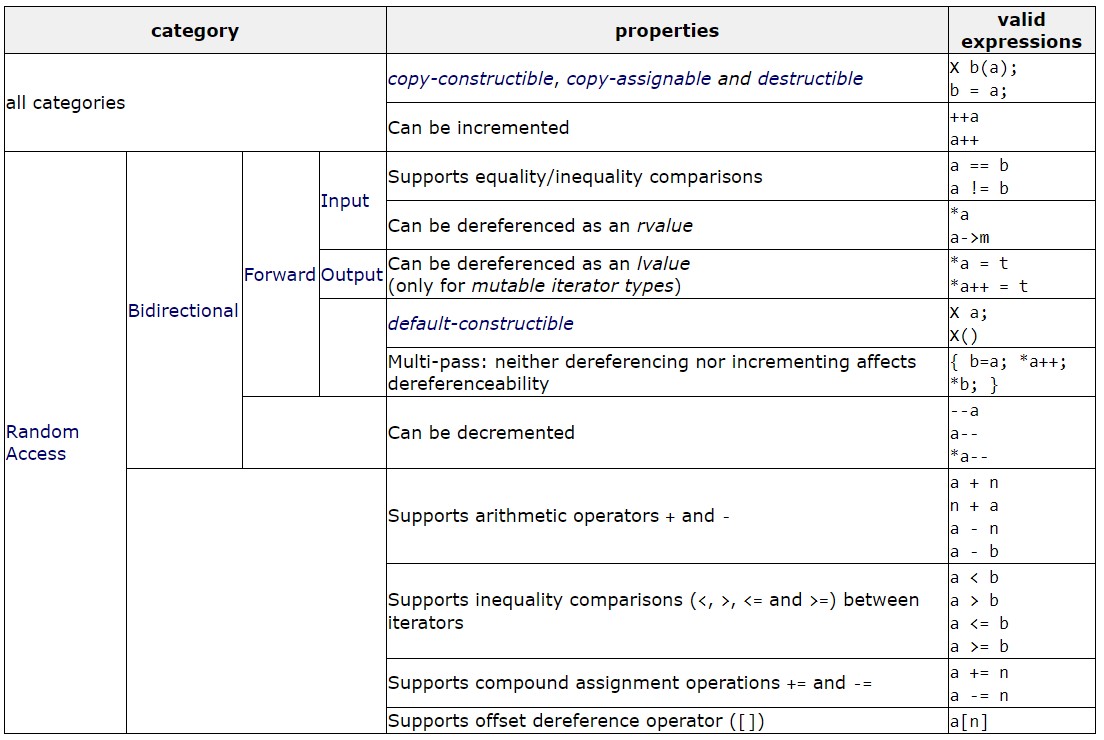




* **Input and output iterators** are the most limited types of iterators:

they can perform sequential single-pass input or output operations.

* All forward, bidirectional and random-access iterators are also valid **input iterators**.
* This arrangement means that a template function which expects for example a **bidirectional** iterator can be provided with a **random access** iterator, but never with a forward iterator.



# != and <

* To write generic code for arbitrary containers, you should use **!=** operator rather than **<** operator. The following loop works with **any container**:
  + for(pos = contner.begin(); pos != contner.end(); ++pos) {...}
* Operator **<** is only provided for random access iterators, so it doesn’t work with lists, sets, and maps. The following does not work with all containers:
  + for(pos = contner.begin(); pos < contner.end(); ++pos) {...}

# The copy Algorithm

## • **copy** (**InputIterator** first, **InputIterator** last, **OutputIterator** result);

* Copies the elements in the range [first,last) into the range beginning at result.
* **copy assumes** that the destination already has room for the elements being copied. It would be an error to copy into an empty list or vector. However, this limitation is easily overcome with insert operators.
* Contained in header file algorithm

#include <iostream>

#include <vector> #include <algorithm> using namespace std;

int main() Output: 0 1 2

{ int arr[] = { 1,2,3 }; vector<int> v(arr, arr + sizeof(arr) / sizeof(arr[0])), v1(3); vector<int>::iterator it; copy(v.begin(), v.end(), v1.begin()+1);

for (it = v1.begin(); it != v1.end(); it++) cout << \*it << " ";

return 0;

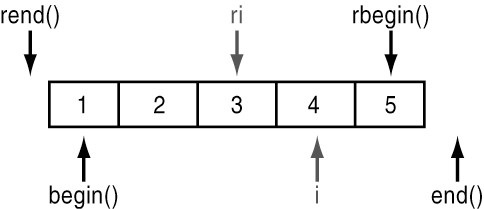
}

# Iterator adapters (Predefined iterators)

1. **Reverse iterators**: operate in reverse
2. **Insert iterators**: allow algorithms to perform in “**insert**” mode rather than “**overwrite**” mode.
3. **Stream iterators**: read and write to I/O stream <http://www.cplusplus.com/reference/iterator/iterator/>

**(1) Reverse iterators**

* **reverse\_iterator:**Used to iterate through the elements of a container in **reverse**
* **const\_reverse\_iterator:Read-only** iterator and is required if container declared as const
* **Obtain reverse iterators** 
  + **rbegin():** returns the reference to the last element in the container
  + **rend():** returns the position **before** the first element in the container



#include <iostream>

|  |
| --- |
| Output: 3 2 1  1 2 3 |

#include <vector> #include <algorithm> using namespace std; int main() { int arr[] = { 1,2,3 }; vector<int> v(arr, arr + sizeof(arr) / sizeof(arr[0])), v2(3); vector<int>::iterator it; vector<int>::reverse\_iterator ir; copy(v.rbegin(), v.rend(), v2.begin());

for (it = v2.begin(); it != v2.end(); it++) cout << \*it << " ";

cout << endl;

for (ir = v2.rbegin(); ir != v2.rend(); ir++) cout << \*ir << " ";

return 0;

}

# (2) Insert iterators

* Insert iterators let you "**point**" to some location in a container and **insert** elements.
* Insert iterators are special output iterators designed to allow algorithms that usually **overwrite** elements (such as copy) to instead **insert** new elements at a specific position in the container.
* The container needs to have an **insert** member function (such as most standard containers).
* Whether value goes **before** or **after** the inserted value depends on what kind of insert operator you've created.
* Many algorithms write to a destination which is assumed to have enough spots. You need to make sure there’s **enough space** in your container.
* Insert operators let the destination container **grow** accordingly.
* You can create an insert iterator with one of the following:

1. **back\_inserter**(container) appends by **push\_back()**
   * vector, deque, list
2. **front\_inserter**(container) insert at the front by **push\_front()**
   * deque, list
3. **Inserter**(container, iterator) insert at iterator pos in the same order by **insert( ).** Constructs an insert iterator that inserts new elements into x in **successive locations** starting at the position pointed by it.

|  |  |
| --- | --- |
| #include <iostream> |  |
| #include <iterator>  #include <vector>  #include <algorithm> | // std::back\_inserter |

|  |
| --- |
| v contains: 3 4 5 30 40 50 |

using namespace std; int main() { vector<int> v, w; for (int i = 3; i <= 5; i++)

{

v.push\_back(i);

w.push\_back(i \* 10); } copy(w.begin(), w.end(), **back\_inserter(v)**);

cout << "v contains: "; for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) cout << \*it <<" ";

return 0;

}

|  |  |
| --- | --- |
| #include <iostream> |  |
| #include <iterator>  #include <deque>  #include <algorithm> | // std::front\_inserter |

|  |
| --- |
| v contains: 50 40 30 3 4 5 |

using namespace std; int main() { **deque**<int> v, w; for (int i = 3; i <= 5; i++)

{

v.push\_back(i);

w.push\_back(i \* 10); } copy(w.begin(), w.end(), **front\_inserter(v)**);

cout << "v contains: "; for (deque<int>::iterator it = v.begin(); it != v.end(); ++it) cout << \*it << " ";

return 0;

}

|  |  |
| --- | --- |
| #include <iostream> |  |
| #include <iterator>  #include <vector>  #include <algorithm> | // std::inserter |

|  |
| --- |
| v contains: 30 40 50 3 4 5 |

using namespace std; int main() { vector<int> v, w; for (int i = 3; i <= 5; i++)

{

v.push\_back(i);

w.push\_back(i \* 10); } copy(w.begin(), w.end(), **inserter(v, v.begin())**);

cout << "v contains: ";

for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) cout << \*it << " ";

return 0;

}

# (3) Stream Iterators

## **(1) istream\_iterator**

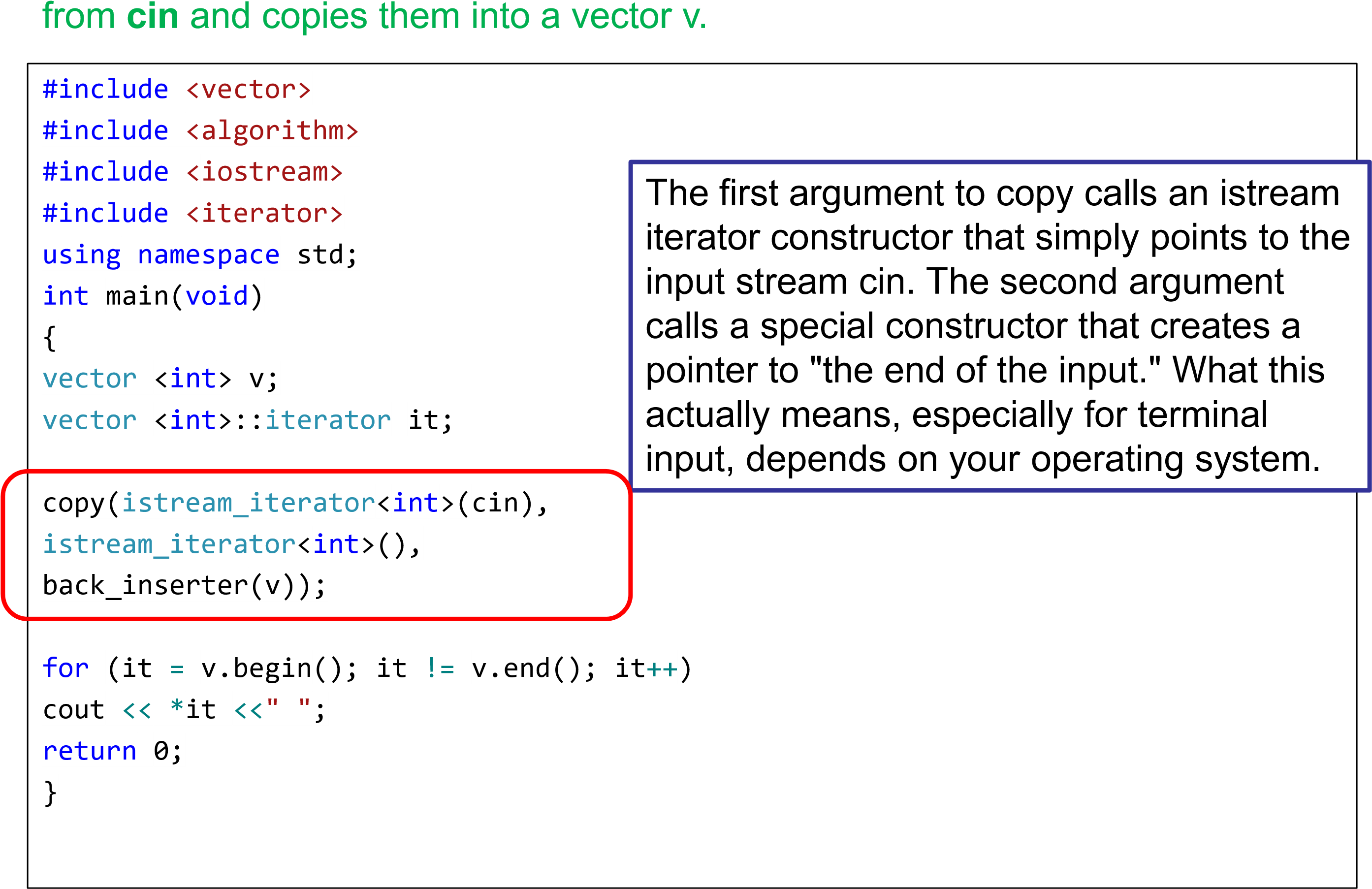
– Used to input data into a program from an input stream

### – class istream\_iterator

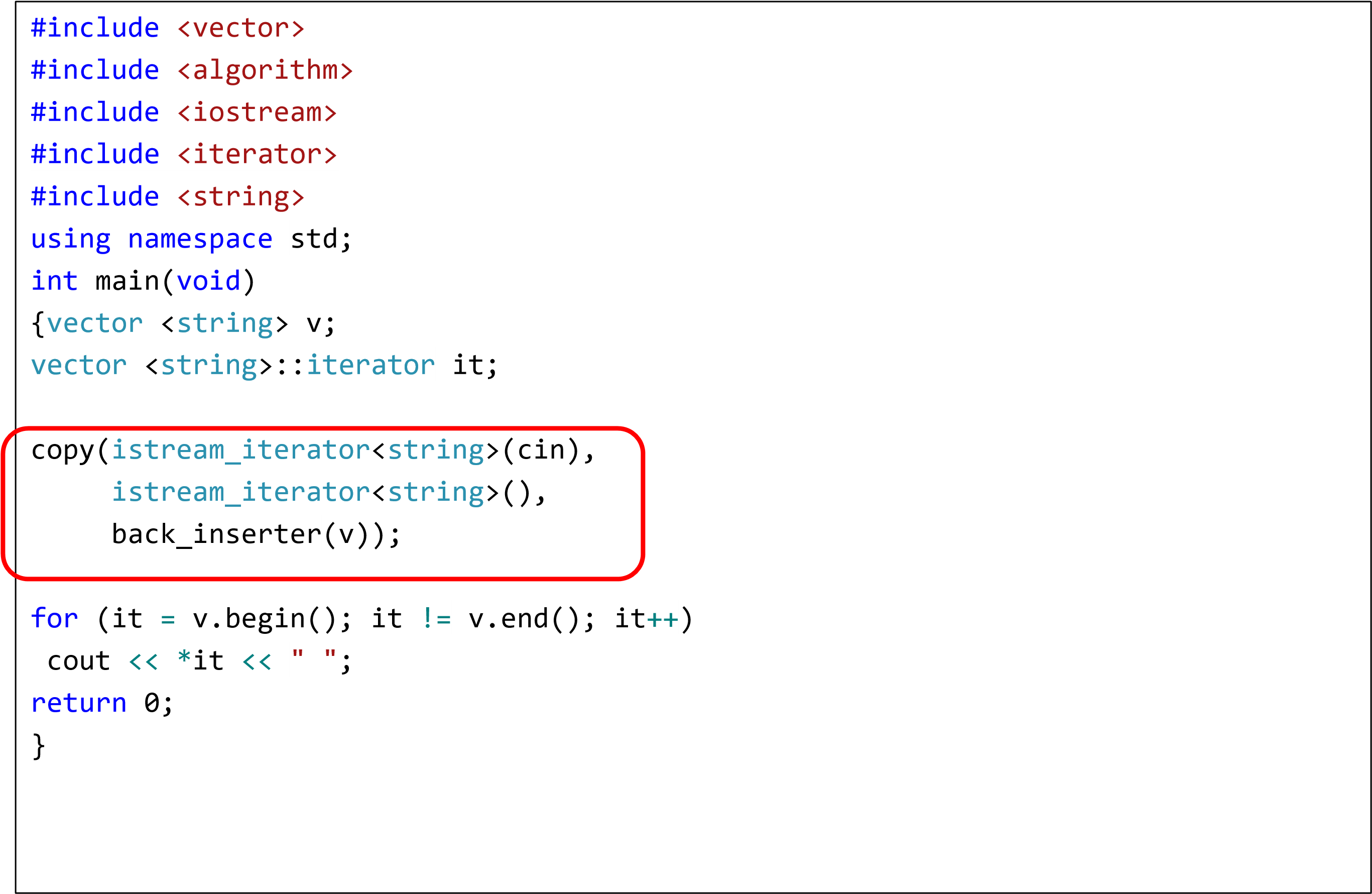
• Contains definition of an input stream iterator – **Istream** iterators are **InputIterators**. – General syntax



The following code fragment constructs an istream iterator that reads **integers**



Read all words from standard input and save into the vector



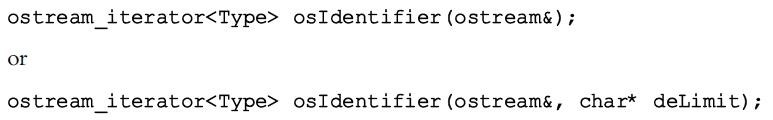
**(2) ostream iterators**

* Used to output data from a program into an output stream

–class ostream\_iterator

• Contains definition of an output stream iterator

* General syntax



* Ostream iterators let you point to an output stream and insert elements into it.
* We can construct an ostream iterator from a C++ output stream as follows:

#include

<

vector

>

#include

>

<

algorithm

#include

<

iostream

>

#include

<

iterator

>

//

std

::

ostream\_iterator

using

namespace

std

;

int

main(

void

)

{

vector

<

int

>

v;

vector

<

int

>::

iterator

Iter1;

for

(

int

i = 0; i < 5; i++)

v.push\_back

(

i

)

;

ostream\_iterator

<

int

>

outIter

(

cout

,

" "

)

;

copy(

v.begin

()

,

v.end

()

,

outIter

)

;

return

0

;

}

**Output:**

4

1 2 3

0

* The first line defines **outIter** to be an ostream iterator for integers. The " " means "put a space between each integer.
* Note how much simpler this is than the equivalent **for** loop with cout and <<.



# pair()

• Treat two values as a single unit

## • #include <**utility**>

* It’s a **struct** type
* All members are public: **first**, **second**
* **pair<type1, type2> nameOfPair;**

## • pair<V1,V2> **make\_pair** (x, y);

– Constructs a pair object with its first element set to x and its second element set to y.

|  |  |
| --- | --- |
| #include <utility>  #include <string> | // std::pair, std::make\_pair |

#include <iostream> The price of lightbulbs is $0.99 using namespace std; The price of shoes is $39.9

The price of tomatoes is $2.3

int main() {

pair <string, double> product1; pair <string, double> product2("tomatoes", 2.30); pair <string, double> product3(product2); product1 = **make\_pair**("lightbulbs", 0.99);

product2.first = "shoes"; product2.second = 39.90;

cout << "The price of " << product1.first << " is $" << product1.second << '\n'; cout << "The price of " << product2.first << " is $" << product2.second << '\n'; cout << "The price of " << product3.first << " is $" << product3.second << '\n'; return 0;

}

• pair<InputIterator1, InputIterator2> **mismatch** (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);

* Compares the elements in the range [first1,last1) with those in the range beginning at first2.
* The function returns a **pair of iterators** to the first element in each range that does not match.

|  |
| --- |
| Mismatching elements: 3 and 5 |

#include <iostream> #include <algorithm>

|  |  |
| --- | --- |
| #include <vector> |  |
| #include <utility> | // std::pair |

using namespace std; int main() { std::vector<int> v1,v2; for (int i = 1; i < 5; i++)

{ v1.push\_back(i);v2.push\_back(i); } v2.at(2) = 5; pair<vector<int>::iterator, vector<int>::iterator> mypair;

mypair = mismatch(v1.begin(), v1.end(), v2.begin()); cout << "Mismatching elements: " << \*mypair.first; cout << " and " << \*mypair.second << '\n'; return 0;}

# **equal\_range**

• Returns the bounds of the subrange that includes all the elements of the range [first,last) with values equivalent to val.

## • pair<ForwardIterator,ForwardIterator> **equal\_range** (ForwardIterator first, ForwardIterator last, const T& val);

• Return value: a **pair** object, whose member

pair::first is an iterator to the lower bound of the subrange of equivalent values, and pair::second its upper bound.

#include <iostream>

|  |
| --- |
| bounds at positions 3 and 6 |

#include <algorithm> #include <vector> using namespace std;

int main() {

int myints[] = { 10,20,30,30,20,10,10,20 }; vector<int> v(myints, myints + 8); // 10 20 30 30 20 10 10 20 pair<vector<int>::iterator, vector<int>::iterator> bounds;

sort(v.begin(), v.end()); // 10 10 10 20 20 20 30 30 bounds = std::equal\_range(v.begin(), v.end(), 20); // ^ ^

cout << "bounds at positions " << (bounds.first - v.begin()); cout << " and " << (bounds.second - v.begin()); return 0;

}

## distance

* Return distance between iterators
* typename

iterator\_traits<InputIterator>::difference\_type

* distance (InputIterator first, InputIterator last);

|  |  |
| --- | --- |
| #include <iostream> | // std::cout |
| #include <iterator> | // std::distance |
| #include <list> | // std::list |

|  |
| --- |
| The distance is: 10 |

int main() { std::list<int> mylist; for (int i = 0; i<10; i++) mylist.push\_back(i \* 10);

std::list<int>::iterator first = mylist.begin(); std::list<int>::iterator last = mylist.end(); std::cout << "The distance is: " << std::**distance**(first, last);

return 0;

}

## Strings

* Strings are objects that represent sequences of characters.
* In C we used **char \*** to represent a string.
* The C++ standard library provides a common implementation of a string class abstraction named **string**.
* To use the string type simply include its header file

#include <string>

* Member functions <http://www.cplusplus.com/reference/string/string/>

## Hello World - C

#include <stdio.h> void main()

{

// create string ‘str’ = “Hello world!” char \*str = “Hello World!”;

printf(“%s\n”, str);

}

## Hello World – C++

#include <iostream> #include <string> using namespace std;

int main()

{

// create string ‘str’ = “Hello world!” string str = "Hello World!";

cout << str << endl; return 0;

}

## String length

• The length of string is returned by its **size()** operation.

|  |
| --- |
| The size of something is 9 characters. |

#include <iostream> #include <string> using namespace std; int main() { string str = "something"; cout << "The size of " << str << " is " << str.size() << " characters."; return 0;

}

## String concatenation

• concatenating one string to another is done by the **‘+’** operator.

Here comes the sun

#include <iostream> #include <string> using namespace std; int main() {

string str1 = "Here "; string str2 = "comes the sun"; string concat\_str = str1 + str2; cout << concat\_str; return 0;

}

## String comparison

• To check if two strings are equal use the ‘**==‘** operator.

string str1 = "Here"; string str2 = "comes the sun"; if (str1 == str2) /\* do something \*/ else

/\* do something else \*/

## String assignment

* To assign one string to another use the “=“ operator.

string str1 = "Hello World"; string str2 = "Hello Dekalb"; str2 = str1;

* Now : str2 equals “Hello World”

## String replace

* Replaces the portion of the string that begins at character pos and spans len characters (or the part of the string in the range between [i1,i2)) by new contents:
* We can use pos or iterator
* <http://www.cplusplus.com/reference/string/string/replace/>

#include <iostream> #include <string> int main()

{ std::string base = "this is a test string."; std::string str2 = "n example"; std::string str3 = "sample phrase"; std::string str4 = "useful.";

// Using positions: 0123456789\*123456789\*12345 std::string str = base; // "this is a test string." str.replace(9, 5, str2); // "this is an example string." (1) str.replace(19, 6, str3, 7, 6); // "this is an example phrase." (2) str.replace(8, 10, "just a"); // "this is just a phrase." (3) str.replace(8, 6, "a shorty", 7); // "this is a short phrase." (4) str.replace(22, 1, 3, '!'); // "this is a short phrase!!!" (5)

// Using iterators: 0123456789\*123456789 str.replace(str.begin(), str.end() - 3, str3); // "sample phrase!!!" str.replace(str.begin(), str.begin() + 6, "replace"); // "replace phrase!!!" str.replace(str.begin() + 8, str.begin() + 14, "is coolness", 7); // "replace is cool!!!" str.replace(str.begin() + 12, str.end() - 4, 4, 'o'); // "replace is cooool!!!" str.replace(str.begin() + 11, str.end(), str4.begin(), str4.end()); // "replace is useful." std::cout << str << '\n'; return 0;

}

## Replace algorithm

* void replace (ForwardIterator first, ForwardIterator last, const T& old\_value, const T& new\_value)
* Assigns new\_value to all the elements in the range [first,last) that compare equal to old\_value.

// replace algorithm example

#include <iostream> // std::cout

#include <algorithm> // std::replace #include <vector> // std::vector

int main() { int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 }; std::vector<int> myvector(myints, myints + 8); std::replace(myvector.begin(), myvector.end(), 20, 99);

std::cout << "myvector contains:";

for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it) std::cout << ' ' << \*it; std::cout << '\n';

return 0;

|  |
| --- |
| myvector contains: 10 99 30 30 99 10 10 99 |

}

// replace algorithm example

#include <iostream> // std::cout #include <algorithm> // std::replace int main() {

std::string s = "thim im a temt mtring."; std::replace(s.begin(), s.end(), 'm', 's');

// replace all 'm' to 's' std::cout << s; return 0;

}

this is a test string.

## Substring

### • string **substr** (pos, len)

* Returns a newly constructed string object with its value initialized to a copy of a substring of this object.
* The substring is the portion of the object that starts at character position pos and spans len characters

(or until the end of the string, whichever comes first).

#include <iostream>

#include <string> #include <algorithm> using namespace std; int main()

{ string str = "The brick walls are not there to keep us out; the brick walls are there to give us a chance to show how badly we want something."; // (quoting Randy Pausch) string str2 = str.substr(4, 5); cout << str2;

return 0;

}

## Visual Studio

* Download Visual Studio Community
  + <https://www.visualstudio.com/vs/community/>
* Microsoft Visual Studio C++ Hello World – YouTube
  + <https://www.youtube.com/watch?v=w49C3SdXshU>
  + [https://msdn.microsoft.com/enus/library/jj620919.aspx](https://msdn.microsoft.com/en-us/library/jj620919.aspx)
* Debugging
  + <https://www.youtube.com/watch?v=z5gBIizwsY0>
  + [http://courses.cs.tamu.edu/choe/12fall/315/lectures/V S-Debugger\_Tutorial.pdf](http://courses.cs.tamu.edu/choe/12fall/315/lectures/VS-Debugger_Tutorial.pdf)

## More STL Algorithms

* void **fill** (iteratorFirst, iteratorLast, val);
  + Assigns val to all the elements in the range [first,last).
* void **fill\_n** (iteratorFirst, n, val);
  + Assigns val to the first n elements of the sequence pointed by first.
* void **generate** (iteratorFirst, iteratorLast, generator);
  + Assigns the value returned by successive calls to generator function to the elements in the range [first,last).
* void **generate\_n** (iteratorFirst, n, generator);
  + Assigns the value returned by successive calls to gen to the first n elements of the sequence pointed by first.
* bool **equal** (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);
  + Compares the elements in the range [first1,last1) with those in the range beginning at first2, and returns true if all of the elements in both ranges match.

#include <iostream>

|  |
| --- |
| myvector contains: 5 5 5 8 8 8 0 0 |

#include <algorithm> #include <vector> using namespace std; int main() { vector<int> myvector(8); vector<int>::iterator it;

fill(myvector.begin(), myvector.begin() + 4, 5); fill(myvector.begin() + 3, myvector.end() - 2, 8);

cout << "myvector contains:";

for (it = myvector.begin(); it != myvector.end(); ++it) cout << ' ' << \*it;

return 0;

}

#include <iostream>

#include <algorithm> myvector contains: 5 5 5 7 7 7 2 2

#include <vector> using namespace std; int main() { vector<int> myvector(8, 2); vector<int>::iterator it;

fill\_n(myvector.begin(), 4, 5); fill\_n(myvector.begin() + 3, 3, 7);

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

cout << " " << \*it;

return 0;

}

#include <iostream>

#include <algorithm>

|  |
| --- |
| myvector contains: 1 2 3 4 5 6 7 8 |

#include <vector> #include <cstdlib> using namespace std; int current = 0; int UniqueNumber() { return ++current; }

int main() { vector<int> myvector(8); vector<int>::iterator it; generate(myvector.begin(), myvector.end(), UniqueNumber);

cout << "myvector contains:"; for (it = myvector.begin(); it != myvector.end(); ++it) cout << ' ' << \*it; return 0;

}

#include <iostream>

#include <algorithm> #include <vector> using namespace std; int main() {

int myints[] = { 20,40,60,80,100 }; vector<int>myvector(myints, myints + 3);

if (equal(myvector.begin(), myvector.end(), myints)) cout << "The contents of both sequences are equal.\n";

else cout << "The contents of both sequences differ.\n";

return 0;

}

* ForwardIterator **remove** (ForwardIterator first,

ForwardIterator last, const T& val);

* + Transforms the range [first,last) into a range with **all** the elements that compare equal to val removed, and returns an iterator to the new end of that range.
  + The function **cannot** alter the **size** of a container.
  + The removal is done by **replacing** the elements that compare equal to val by the next element that does not, and signaling the new size of the shortened range by returning an iterator to the element that should be considered its new past-the-end element.
* ForwardIterator **remove\_if** (ForwardIterator first,

ForwardIterator last, UnaryPredicate **pred**);

* + Transforms the range [first,last) into a range with all the elements for which pred returns true removed, and returns an iterator to the new end of that range.

### – **pred**

• Unary ***function*** that accepts an element in the range as argument, and returns a value convertible to bool. The value returned indicates whether the element is to be removed (if true, it is removed).

#include <iostream>

#include <algorithm> 4 5

#include <vector> using namespace std; int main() {

vector<int> v; vector<int>::iterator it, it2; v.push\_back(2);v.push\_back(4);

v.push\_back(2);v.push\_back(5); it2= remove(v.begin(), v.end(), 2);

for (it = v.begin(); it != it2; ++it) cout << \*it <<" "; return 0;}

#include <iostream>

#include <algorithm> // std::remove\_if 2 4 6

#include <vector> using namespace std;

bool IsOdd(int i) { return ((i % 2) == 1); } int main() { vector<int> v; vector<int>::iterator it, it2; v.push\_back(2);v.push\_back(3);

v.push\_back(4);v.push\_back(5);

v.push\_back(6);v.push\_back(7);

it2

=

remove\_if

(

v.begin

()

,

v.end

()

,

IsOdd

)

;

for (it = v.begin(); it != it2; ++it) cout << \*it << " ";

return 0;

}

## Array manipulation

* void **reverse** (BidirectionalIterator first, BidirectionalIterator last);
  + Reverses the order of the elements in the range [first,last).
* void **rotate** (ForwardIterator first, ForwardIterator middle,

ForwardIterator last);

* + Rotates the order of the elements in the range [first,last), in such a way that the element pointed by middle becomes the new first element.

#include <iostream>

#include <algorithm> // std::reverse

#include <vector>

int main() { std::vector<int> myvector;

9 8 7 6 5 4 3 2 1

for (int i = 1; i<10; ++i) myvector.push\_back(i); std::**reverse**(myvector.begin(), myvector.end());

std::cout << "myvector contains:"; for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it) std::cout << ' ' << \*it; std::cout << '\n';

return 0;

}

#include <iostream>

4 5 6 7 8 9 1 2 3

#include <algorithm> // std::rotate

#include <vector>

int main() {

std::vector<int> myvector; for (int i = 1; i<10; ++i) myvector.push\_back(i);

std::**rotate**(myvector.begin(), myvector.begin() + 3, myvector.end());

std::cout << "myvector contains:"; for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it) std::cout << ' ' << \*it; std::cout << '\n';

return 0;

}

## Mathematical Algorithms

* X **count** (InputIterator first, InputIterator last, const T& val);
  + Returns the number of elements (X) in the range [first,last) that compare equal to val.
* X **count\_if** (InputIterator first, InputIterator last,

UnaryPredicate pred);

* + Returns the number of elements in the range [first,last) for which pred is true.
* Function **for\_each** (InputIterator first, InputIterator last,

Function fn);

* + Applies function fn to each of the elements in the range

[first,last).

* OutputIterator **transform** (InputIterator first, InputIterator last,

OutputIterator result, UnaryOperation op);

* + Applies an operation sequentially to the elements in the range

[first,last) and **stores** the result in the range that begins at result.

* const T& **max** (const T& a, const T& b); – Returns the largest of a and b.
* const T& **min** (const T& a, const T& b); – Returns the smallest of a and b.
* ForwardIterator **min\_element** (ForwardIterator first, ForwardIterator last);
  + Returns an **iterator** pointing to the element with the smallest value in the range [first,last).
* ForwardIterator **max\_element** (ForwardIterator first, ForwardIterator last);
  + Returns an **iterator** pointing to the element with the largest value in the range [first,last).
* void **random\_shuffle** (RandomAccessIterator first, RandomAccessIterator last);
  + Rearranges the elements in the range [first,last) randomly.
* **accumulate** (InputIterator first, InputIterator last, T init);
  + Returns the result of accumulating all the values in the range

[first,last) to init.

|  |
| --- |
| 3  2  1  12  100 4 64 1 81 9 64 64 81 144  11 3 9 2 10 4 9 9 10 13 |

#include <iostream>

#include <algorithm>

#include <vector> #include <iterator> using namespace std;

bool fun1(int value) {return value > 9;} void fun2(int value) {cout << value \* value << ' ';} int fun3(int value) {return ++value;}

int main()

{ostream\_iterator< int > output(cout, " "); int a[10] = { 10, 2, 8, 1, 9, 3, 8, 8, 9, 12 }; vector<int> v(a, a + 10);

cout << **count**(v.begin(), v.end(), 8) << endl; cout << **count\_if**(v.begin(), v.end(), fun1) << endl; cout << \*(**min\_element**(v.begin(), v.end())) << endl; cout << \*(**max\_element**(v.begin(), v.end())) << endl; **for\_each**(v.begin(), v.end(), fun2); cout << endl; **transform**(v.begin(), v.end(), **v.begin()**, fun3); copy(v.begin(), v.end(), output); return 0;}

#include <iostream>

|  |  |
| --- | --- |
| #include <numeric> | // std::accumulate |

int main() { using default accumulate: 200 int init = 100;

int numbers[] = { 10,20,30,40 };

std::cout << "using default accumulate: ";

std::cout << std::accumulate(numbers, numbers + 4, init); std::cout << '\n';

return 0;

}

## Exercise

• Use the STL to write a C++ program that uses a vector to store the alphabet using the generate function and then prints it to the screen using the copy function.

#include <iostream>

#include <algorithm>

#include <vector> #include <iterator> using namespace std;

char nextLetter() {static char letter = 'A'; return letter++;} int main() { vector< char > chars(26); ostream\_iterator< char > output(cout, " "); generate(chars.begin(), chars.end(), nextLetter); copy(chars.begin(), chars.end(), output);

return 0;

}

### Container Adapters

* Classes that use an encapsulated object of a specific container class as its **underlying** container, providing a specific set of member functions to access its elements.
* **Container adapters** are **not first-class containers,** because
  + they do not provide the actual **data-structure** implementation in which elements can be stored.
  + they do not support **iterators**.

### (1) stack

* Class stack enables insertions into and deletions at **one end** (**last-in, first-out (LIFO)** data structure).
* The underlying container shall support the following operations: empty, size, back, push\_back, and pop\_back
* A stack can be implemented with any of the sequence containers: vector, list and deque.
  + // stack with **default** underlying deque
  + stack<int> intDequeStack;
  + // stack with underlying vector
  + stack< int, vector<int>> intVectorStack;
  + // stack with underlying list
  + stack< int, list<int >> intListStack;

|  |  |  |
| --- | --- | --- |
| **stack operations** | **Description** | **Implemented by calling function** |
| **push** | Insert an element at the top of the stack | push\_back |
| **pop** | Remove the top element of the stack | pop\_back |
| **top** | Get a reference to the top element of the stack | back |
| **empty** | Determine whether the stack is empty | empty |
| **size** | Get the number of elements in the stack | size |

* Header **<stack>** must be included to use class stack
* void **push** (const value\_type& val);
* void **pop**();
* value\_type& **top**();
* bool **empty**() const;
* size\_type **size**() const;

Write a C++ program that initializes the content of a stack to a sequence of numbers (from 1 to 5) and then pops the elements one by one until it is empty and calculates their sum.

#include <iostream> #include <stack> using namespace std; int main()

{ stack<int> mystack; int sum=0;

for (int i = 1;i <= 5;i++) mystack.push(i);

while (!mystack.empty())

{ sum += mystack.top(); mystack.pop();

} cout << "total: " << sum ;

return 0;}

* A guest indicates that he can’t eat mushrooms and needs to have them removed.



* Write a function that accepts a reference to a stack ***s*** and an integer ***target*** and returns true if target was removed and false otherwise. You can only use stack data structure and operations. The order of the remaining items should be same as the initial stack s.

#include <iostream> #include <stack> using namespace std; bool removeTarget(stack<int> &s, int target)

{ stack<int> tempStk; bool foundTarget = true;

while (!s.empty() && s.top() != target)

{tempStk.push(s.top());

s.pop();}

// if s is not empty then we found the target if (!s.empty())

s.pop();

else foundTarget = false;

// Restore the items while (!tempStk.empty())

{s.push(tempStk.top()); tempStk.pop();}

return foundTarget;

} int main()

{stack<int> mystack; int sum = 0; for (int i = 1;i <= 5;i++) mystack.push(i); if(removeTarget(mystack, 3)) while (!mystack.empty()) { cout<< mystack.top() <<" "; mystack.pop(); } return 0;}

bool removeTarget(stack<int> &s, int y)

{ stack<int> s2; bool flag = false; int x;

while (!s.empty() && !flag)

{ x = s.top(); if (x == y)

flag = true;

else s2.push(x);

s.pop(); }

if (flag)

{ while (!s2.empty())

{ x = s2.top(); s.push(x); s2.pop();

} return true; } else return false;

}

|  |
| --- |
| 10  10  8 |

#include <iostream> #include <stack> using namespace std; int main() {stack<int> s; int x = 3, y = 5; s.push(8);

s.push(x - 1);

s.push(y \* 2); x = s.top(); cout << x << endl; // Output 1: \_\_\_\_\_\_\_\_\_\_\_\_\_ y = s.top();

s.pop(); cout << y << endl; // Output 2: \_\_\_\_\_\_\_\_\_\_\_\_\_ s.pop(); while (!s.empty())

{ y = s.top();

s.pop(); cout << y; // Output 3: \_\_\_\_\_\_\_\_\_\_\_\_\_

}return 0;}

### (2) queue

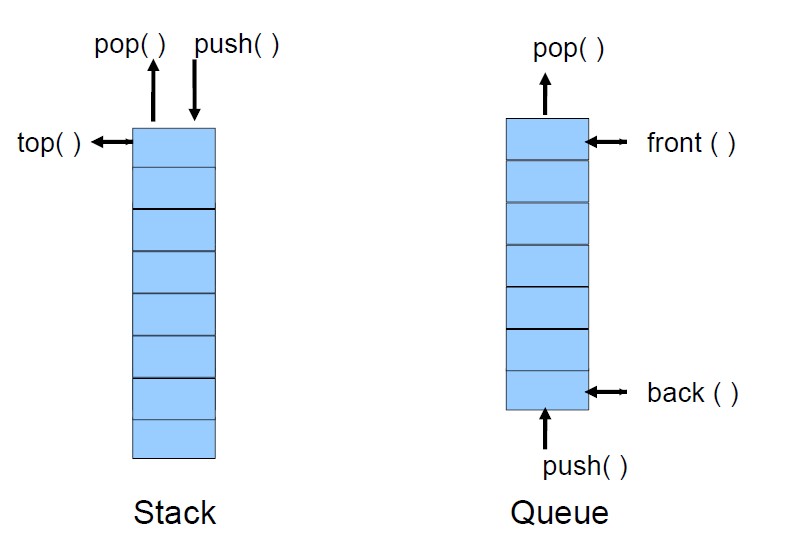
* Class queue enables **insertions at the back** and **deletions from the front** (i.e. **oldest element**)

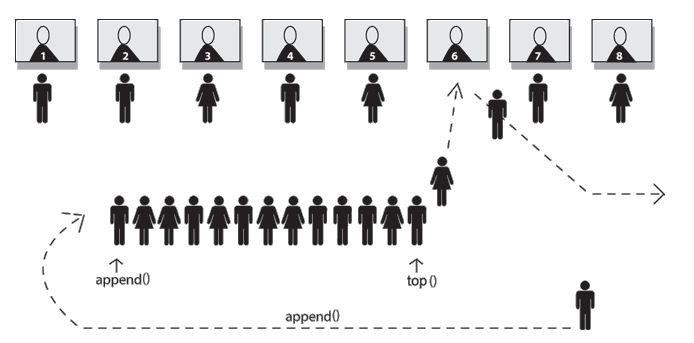
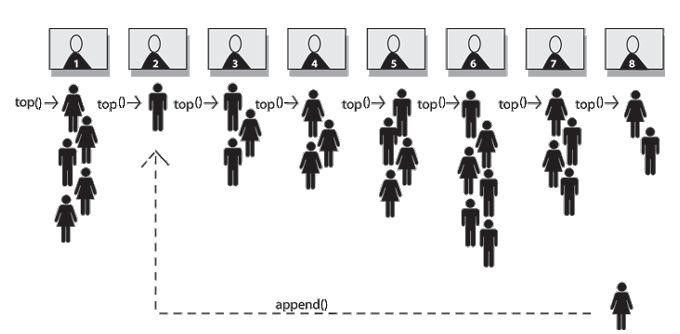
(**first-in, first-out (FIFO)** data structure)

* A queue can be implemented with STL data structure **list** or **deque**.
* By default, a queue is implemented with a **deque**.
* Header <queue> must be included to use a queue.
* Applications
  + Printer
  + <https://en.wikipedia.org/wiki/Queueing_theory>

|  |  |  |
| --- | --- | --- |
| **Queue operations** | **Description** | **Implemented by calling function** |
| **push** | Insert an element at the back of the queue | push\_back |
| **pop** | Remove the element at the front of the queue | pop\_front |
| **front** | Get a reference to the first element in the queue | front |
| **back** | Get a reference to the last element in the queue | back |
| **empty** | Determine whether the queue is empty | empty |
| **size** | Get the number of elements in the queue | size |

* void **push** (const value\_type& val);
* void **pop**();
* value\_type& **front**();
* value\_type& **back**();
* bool **empty**();
* size\_type **size()**;





Complete the following code so that even numbers will be added to q1 and odd numbers to q2.

queue<double> q1, q2; int value;

cout << "\n Enter an integer > 0 or 0 to quit:"; cin >> value; while (value != 0)

{ cout << "\n Enter an integer > 0 or 0 to quit:"; cin >> value;

}

queue<double> q1, q2; int value;

cout << "\n Enter an integer > 0 or 0 to quit:"; cin >> value;

while (value != 0)

{ if (value % 2 == 0) q1.push(value);

else q2.push(value);

cout << "\n Enter an integer > 0 or 0 to quit:"; cin >> value;

}

# **(3) priority\_queue**

* Provides functionality that enables **insertions in sorted order** and **deletions from the front**.
* When elements are added to a priority\_queue, they’re inserted in priority order, such that the **highest-priority element** (i.e., the largest value) will be the first element removed from the priority\_queue.
* Priority ranking instead of chronological order.
* Can be implemented with STL sequence containers **vector** or **deque**.
* By default, a priority\_queue is implemented with a **vector**.
* Header **<queue>** must be included to use class priority\_queue
* Examples
  + Operating system scheduler
  + Patients in an emergency room

|  |  |  |
| --- | --- | --- |
| **Queue operations** | **Description** | **Implemented by calling function** |
| **push** | Inserts an element at the appropriate location based on priority order of the  priority\_queue | push\_back, then reordering the elements using heapsort |
| **pop** | removes the highest-priority element | pop\_back |
| **top** | gets a reference to the top element | front |
| **empty** |  | empty |
| **size** |  | size |

#include <iostream>

#include <queue> 9.2, 5.5, 2.5, 1.3,

using namespace std;

int main()

{ priority\_queue<double> mypq;

mypq.push(2.5); mypq.push(9.2); mypq.push(1.3); mypq.push(5.5);

while (!mypq.empty())

{ cout << mypq.top() << ", "; mypq.pop();

} return 0;

}

* What is a stack?
  + [Apple](http://www.glassdoor.com/Interview/What-is-a-stack-QTN_50744.htm) [http://www.glassdoor.com/Interview/What-is-a-stackQTN\_50744.htm](http://www.glassdoor.com/Interview/What-is-a-stack-QTN_50744.htm)
* What is the difference between a queue and a stack?
  + [CISCO http://www.glassdoor.com/Interview/What-is-thedifference-between-a-queue-and-a-stack-Then-describe-which-](http://www.glassdoor.com/Interview/What-is-the-difference-between-a-queue-and-a-stack-Then-describe-which-one-should-be-implemented-with-a-singly-linked-li-QTN_276015.htm)

[one-should-be-implemented-with-a-singly-linked-liQTN\_276015.htm](http://www.glassdoor.com/Interview/What-is-the-difference-between-a-queue-and-a-stack-Then-describe-which-one-should-be-implemented-with-a-singly-linked-li-QTN_276015.htm)

* Push values 1, 2, 3, 4 and 5 onto a stack s and then print them in reverse order.
* DualStack
  + Which STL container can be used to store two stacks? – Which member functions would you need?
* Implement a queue with two stacks.
  + [Google http://www.glassdoor.com/Interview/How-to-implement-a-queuesimply-using-two-stacks-and-how-to-implement-a-highly-efficient-queueusing-two-stacks-QTN\_41158.htm](http://www.glassdoor.com/Interview/How-to-implement-a-queue-simply-using-two-stacks-and-how-to-implement-a-highly-efficient-queue-using-two-stacks-QTN_41158.htm)
  + [Facebook http://www.glassdoor.com/Interview/How-can-one-implementa-queue-with-only-a-stack-implementation-QTN\_200453.htm](http://www.glassdoor.com/Interview/How-can-one-implement-a-queue-with-only-a-stack-implementation-QTN_200453.htm)
  + [Amazon](http://www.glassdoor.com/Interview/Create-a-Queue-using-two-Stacks-QTN_249668.htm) [http://www.glassdoor.com/Interview/Create-a-Queue-usingtwo-Stacks-QTN\_249668.htm](http://www.glassdoor.com/Interview/Create-a-Queue-using-two-Stacks-QTN_249668.htm)
* Implement a stack using two queues.
  + [Microsoft](http://www.glassdoor.com/Interview/Implement-a-stack-using-two-queues-QTN_183864.htm) [http://www.glassdoor.com/Interview/Implement-a-stack-usingtwo-queues-QTN\_183864.htm](http://www.glassdoor.com/Interview/Implement-a-stack-using-two-queues-QTN_183864.htm)
* Implement a priority stack with two stacks
  + [Bloomberg http://www.glassdoor.com/Interview/How-to-implement-apriority-stack-with-two-stacks-QTN\_137146.htm](http://www.glassdoor.com/Interview/How-to-implement-a-priority-stack-with-two-stacks-QTN_137146.htm)