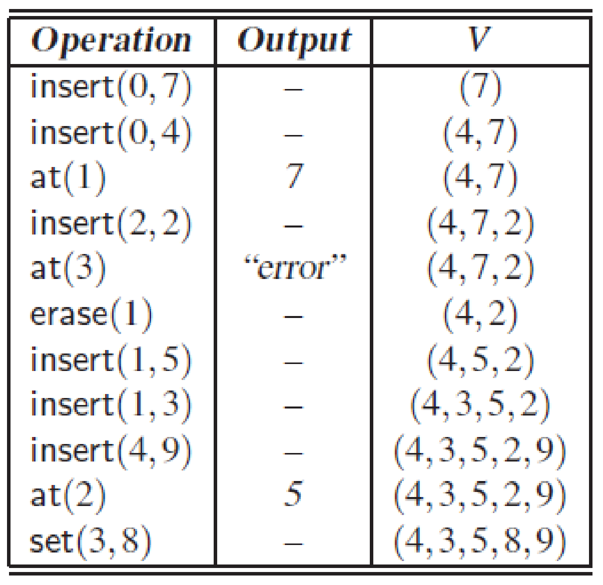
CH6. LIST AND ITERATOR ADTS

CSED233 Data Structure
Prof. Hwanjo Yu
POSTECH

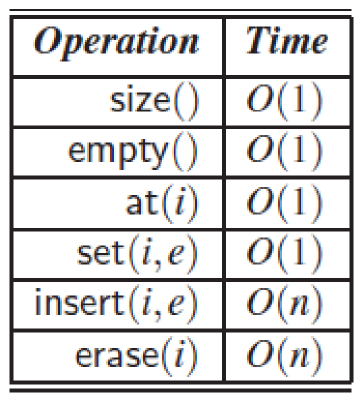
Vectors (§ 6.1)

- Vector
  - *Member functions*
    - at(i)
    - set(i,e)
    - insert(i,e)
    - erase(i)

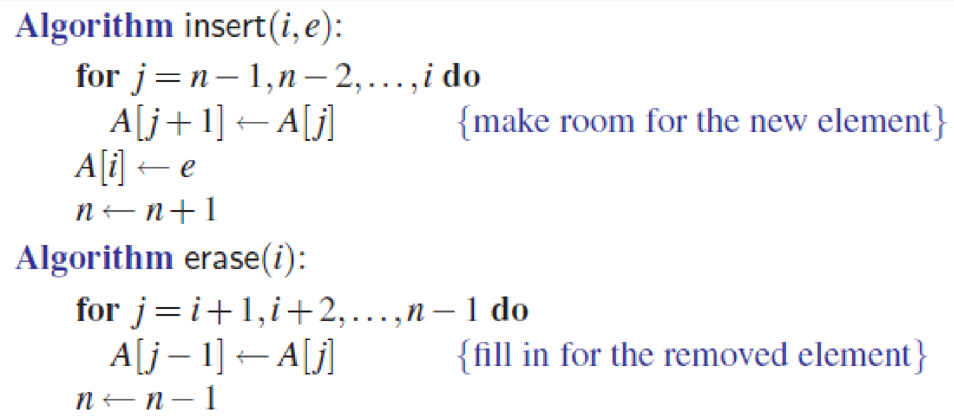


Operation	*Output*	V
insert$(0,7)$	–	(7)
insert$(0,4)$	–	$(4,7)$
at(1)	7	$(4,7)$
insert$(2,2)$	–	$(4,7,2)$
at(3)	"error"	$(4,7,2)$
erase(1)	–	$(4,2)$
insert$(1,5)$	–	$(4,5,2)$
insert$(1,3)$	–	$(4,3,5,2)$
insert$(4,9)$	–	$(4,3,5,2,9)$
at(2)	5	$(4,3,5,2,9)$
set$(3,8)$	–	$(4,3,5,8,9)$

- Array-based implementation



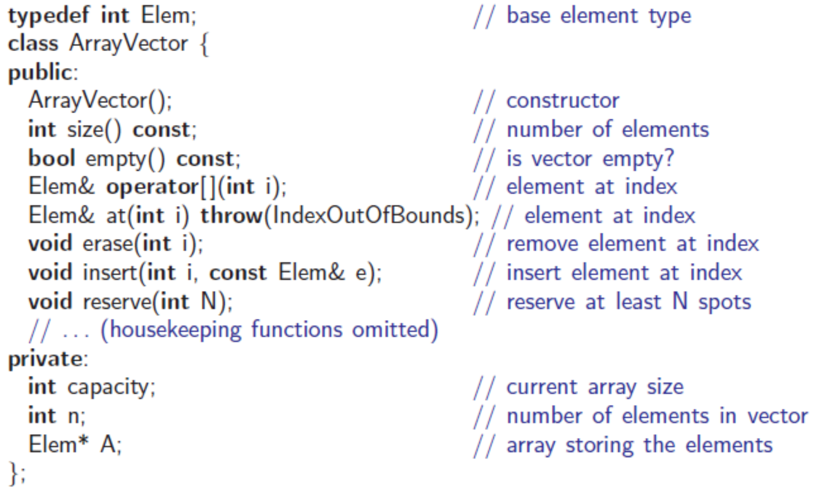
Operation	*Time*
size()	$O(1)$
empty()	$O(1)$
at(i)	$O(1)$
set(i,e)	$O(1)$
insert(i,e)	$O(n)$
erase(i)	$O(n)$



```
Algorithm insert($i,e$):
    for $j = n-1, n-2, \ldots, i$ do
        $A[j+1] \leftarrow A[j]$          {make room for the new element}
    $A[i] \leftarrow e$
    $n \leftarrow n+1$
Algorithm erase($i$):
    for $j = i+1, i+2, \ldots, n-1$ do
        $A[j-1] \leftarrow A[j]$          {fill in for the removed element}
    $n \leftarrow n-1$
```

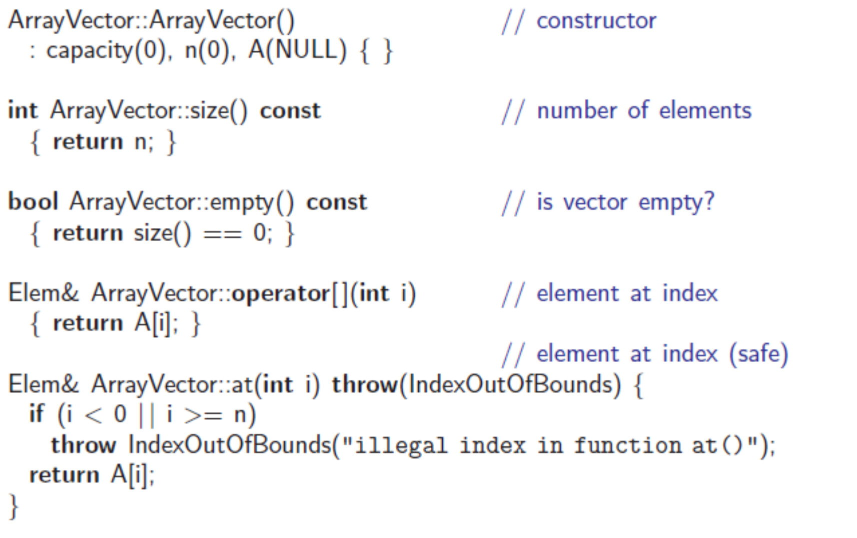
Extendable Array (§ 6.1.3)

- When an overflow occurs
  - *Allocate a new array B of capacity N*
  - *Copy A[i] to B[i], for i=0,…,N-1*
  - *Deallocate A and reassign A to point to the new array B*
- v.set(i,5) can be implemented either
  - *v[i]=5*
  - *v.at(i)=5*



```
typedef int Elem;                                  // base element type
class ArrayVector {
public:
  ArrayVector();                                   // constructor
  int size() const;                                // number of elements
  bool empty() const;                              // is vector empty?
  Elem& operator[](int i);                         // element at index
  Elem& at(int i) throw(IndexOutOfBounds);         // element at index
  void erase(int i);                               // remove element at index
  void insert(int i, const Elem& e);               // insert element at index
  void reserve(int N);                             // reserve at least N spots
  // . . . (housekeeping functions omitted)
private:
  int capacity;                                    // current array size
  int n;                                           // number of elements in vector
  Elem* A;                                         // array storing the elements
};
```

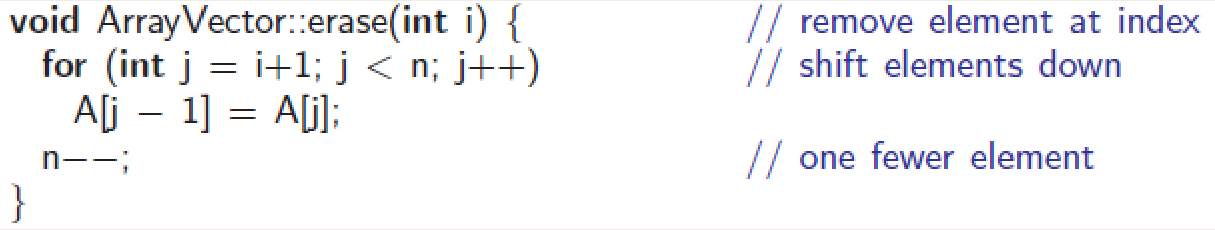
Code Fragment 6.2: A vector implementation using an extendable array.



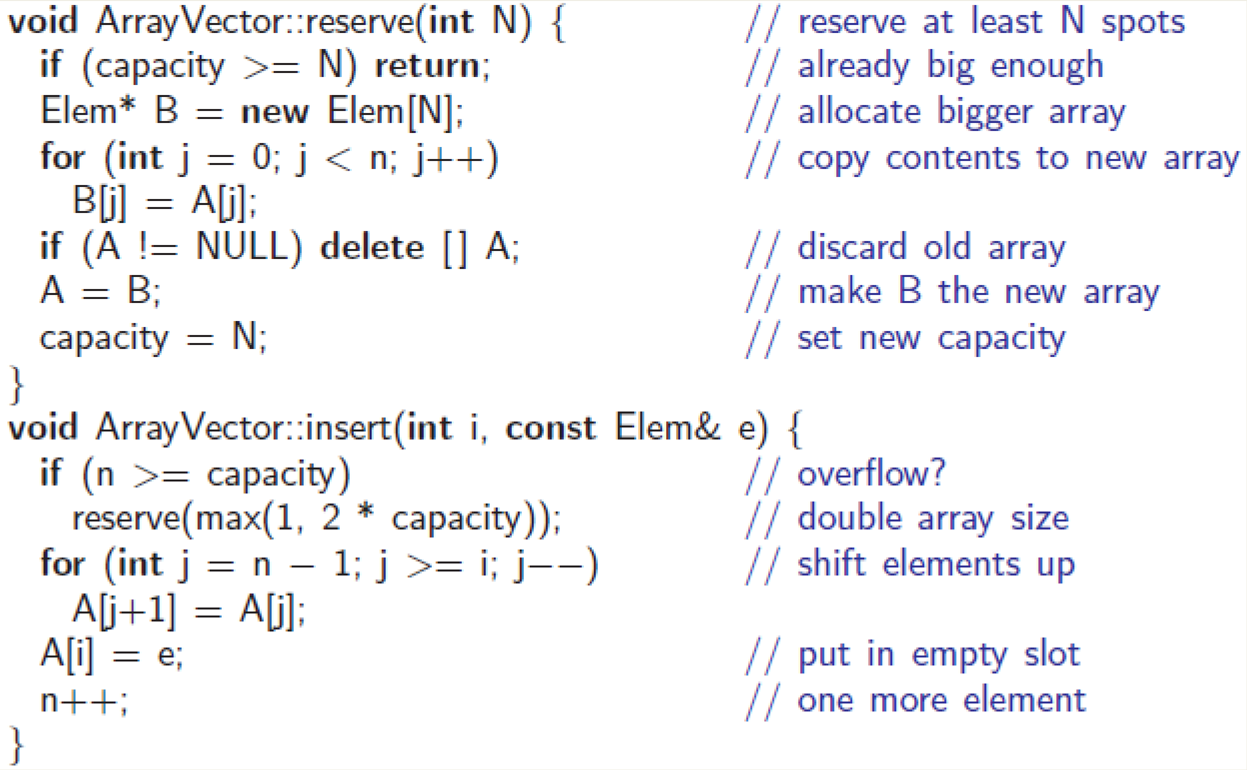
```
ArrayVector::ArrayVector()                         // constructor
  : capacity(0), n(0), A(NULL) { }

int ArrayVector::size() const                      // number of elements
  { return n; }

bool ArrayVector::empty() const                    // is vector empty?
  { return size() == 0; }

Elem& ArrayVector::operator[](int i)               // element at index
  { return A[i]; }
                                                   // element at index (safe)
Elem& ArrayVector::at(int i) throw(IndexOutOfBounds) {
  if (i < 0 || i >= n)
    throw IndexOutOfBounds("illegal index in function at()");
  return A[i];
}
```

Code Fragment 6.3: The simple member functions for class ArrayVector.

Extendable Array (§ 6.1.3)



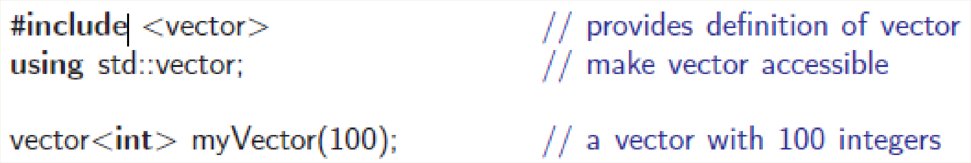
```
void ArrayVector::erase(int i) {            // remove element at index
  for (int j = i+1; j < n; j++)             // shift elements down
    A[j − 1] = A[j];
  n−−;                                      // one fewer element
}
```



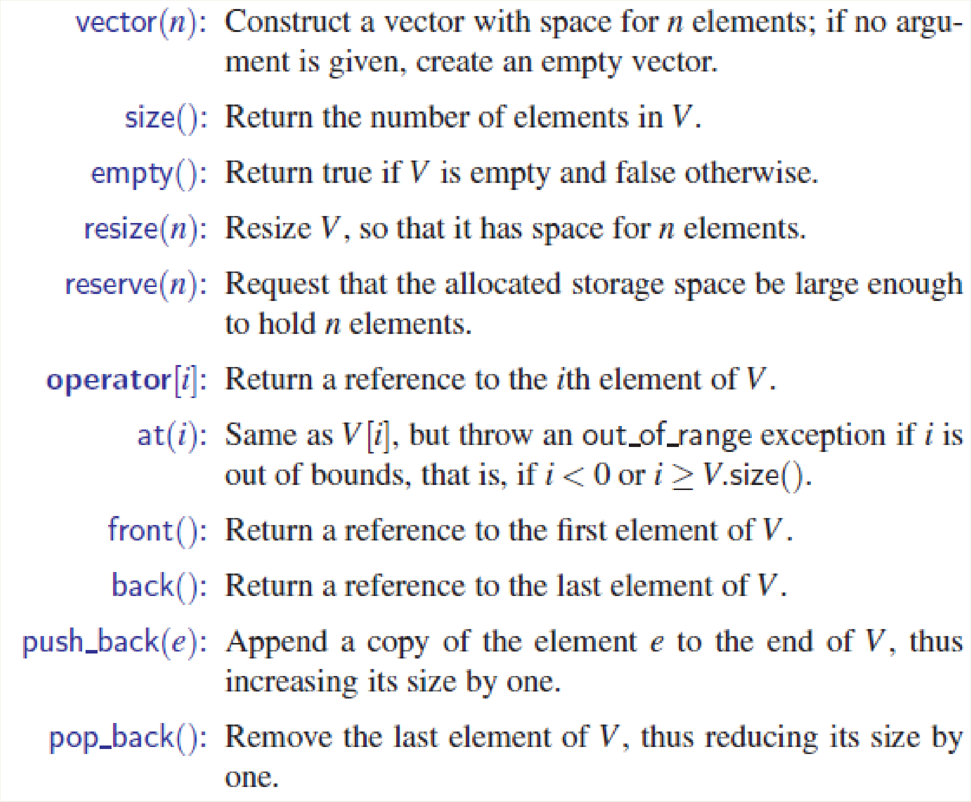
```
void ArrayVector::reserve(int N) {          // reserve at least N spots
  if (capacity >= N) return;                // already big enough
  Elem* B = new Elem[N];                    // allocate bigger array
  for (int j = 0; j < n; j++)               // copy contents to new array
    B[j] = A[j];
  if (A != NULL) delete [] A;               // discard old array
  A = B;                                    // make B the new array
  capacity = N;                             // set new capacity
}
void ArrayVector::insert(int i, const Elem& e) {
  if (n >= capacity)                        // overflow?
    reserve(max(1, 2 * capacity));          // double array size
  for (int j = n − 1; j >= i; j−−)          // shift elements up
    A[j+1] = A[j];
  A[i] = e;                                 // put in empty slot
  n++;                                      // one more element
}
```

STL Vectors (§ 6.1.4)

- One of STL sequence containers (stacks, queues, lists, etc.)



```
#include <vector>                    // provides definition of vector
using std::vector;                   // make vector accessible

vector<int> myVector(100);           // a vector with 100 integers
```



vector(n): Construct a vector with space for n elements; if no argument is given, create an empty vector.

size(): Return the number of elements in V.

empty(): Return true if V is empty and false otherwise.

resize(n): Resize V, so that it has space for n elements.

reserve(n): Request that the allocated storage space be large enough to hold n elements.

operator[i]: Return a reference to the ith element of V.

at(i): Same as $V[i]$, but throw an out_of_range exception if i is out of bounds, that is, if $i < 0$ or $i \geq V.\text{size}()$.

front(): Return a reference to the first element of V.

back(): Return a reference to the last element of V.

push_back(e): Append a copy of the element e to the end of V, thus increasing its size by one.

pop_back(): Remove the last element of V, thus reducing its size by one.

Lists (§ 6.2)

- Lists
  - *Accessing an element with its index is O(n). Why?*
  - *Insert(v,e), which inserts e before v, is O(1)*
  - *But, node-based operations could be dangerous by letting a user to modify the internal structure of a list*
  - *Thus, hide pointers to the user. How? => Containers and Iterators*
- Containers
  - *Sequences: vector, deque, list*
  - *Associative containers: set (multiset), map (multimap)*
- Iterators
  - *Iterator p = L.begin() or p = L.end()*
  - *Operator overloading: ++p, - -p, *p*

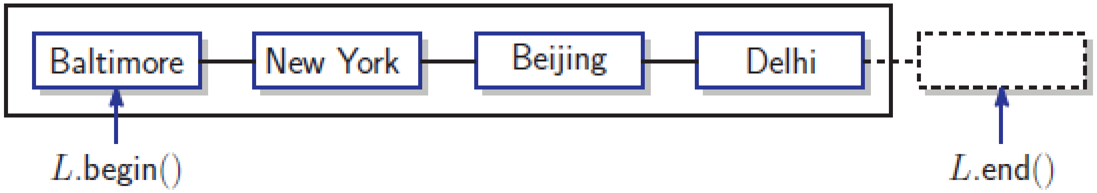
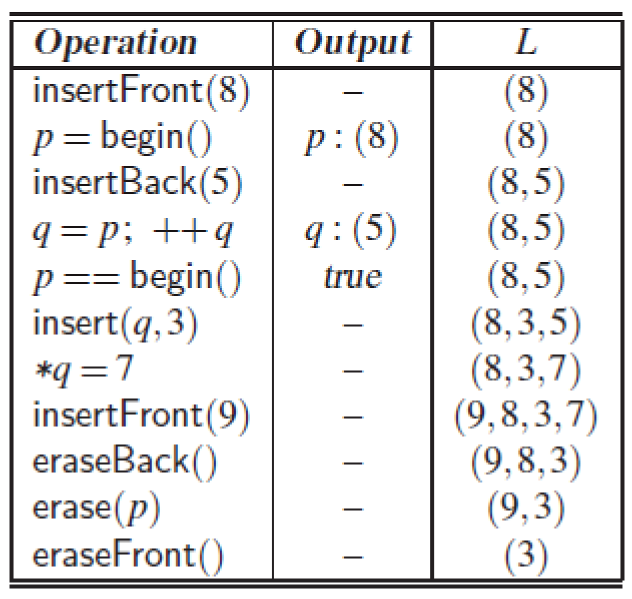


Figure 6.5: The special iterators L.begin() and L.end() for a list L.

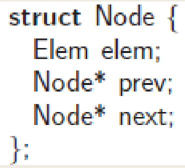
List ADT (§ 6.2.2)

- Member functions
  - *begin(): return an iterator*
  - *end(): return an iterator*
  - *insert(p,e): insert e before position p (iterator)*
  - *insertFront(e): insert(L.begin(), e)*
  - *insertBack(e): insert(L.end(), e)*
  - *eraseFront()*
  - *eraseBack()*
  - *erase(p): delete the element at position p (iterator)*

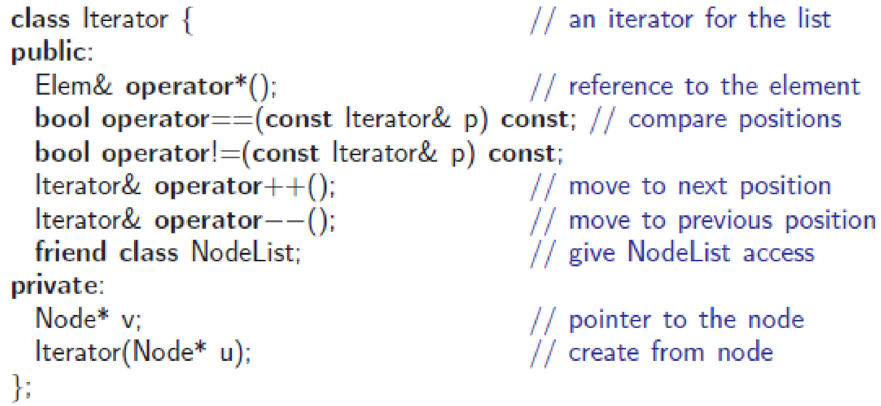


Operation	**Output**	L
insertFront(8)	–	(8)
$p =$ begin()	$p : (8)$	(8)
insertBack(5)	–	(8,5)
$q = p;\ ++q$	$q : (5)$	(8,5)
$p ==$ begin()	*true*	(8,5)
insert(q,3)	–	(8,3,5)
$*q = 7$	–	(8,3,7)
insertFront(9)	–	(9,8,3,7)
eraseBack()	–	(9,8,3)
erase(p)	–	(9,3)
eraseFront()	–	(3)

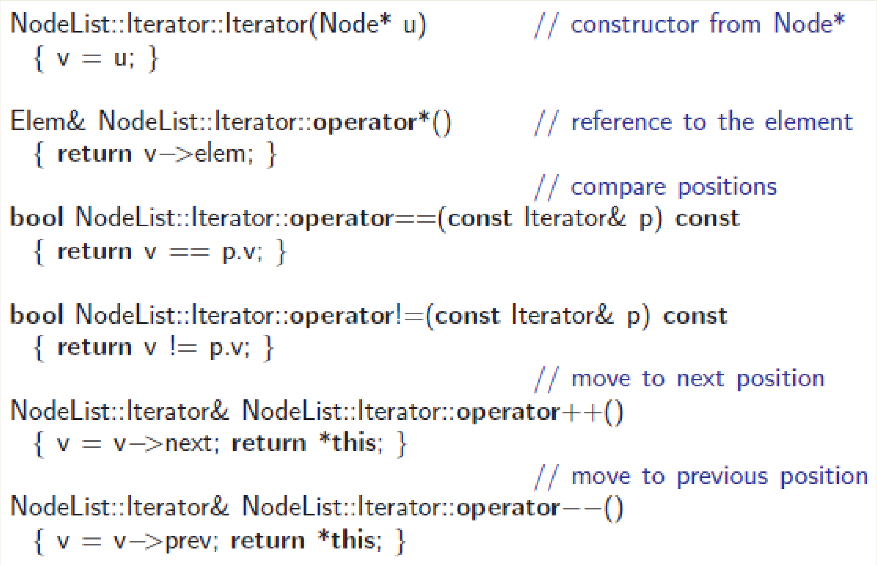
Doubly Linked List Implementation (§ 6.2.3)



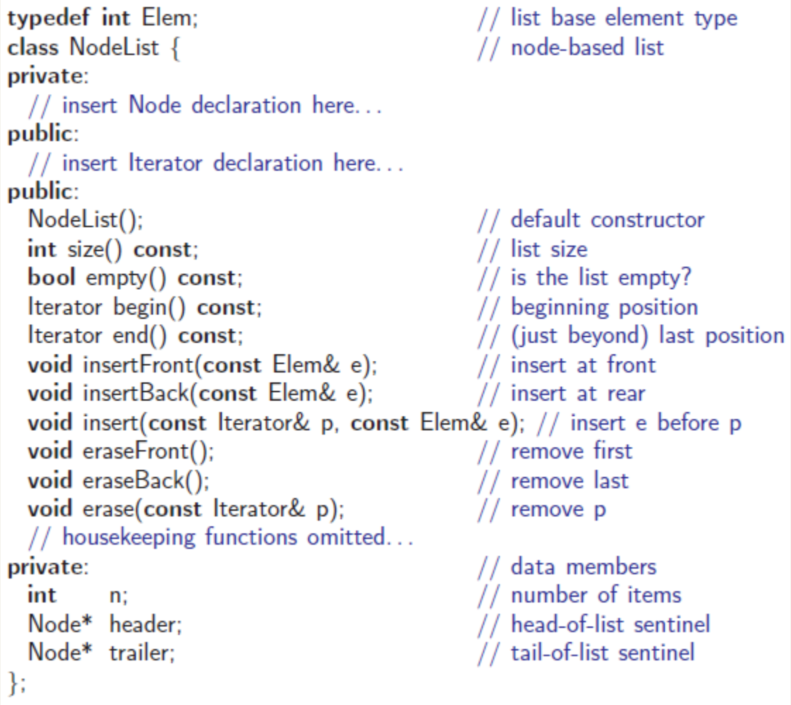
```
struct Node {
  Elem elem;
  Node* prev;
  Node* next;
};
```



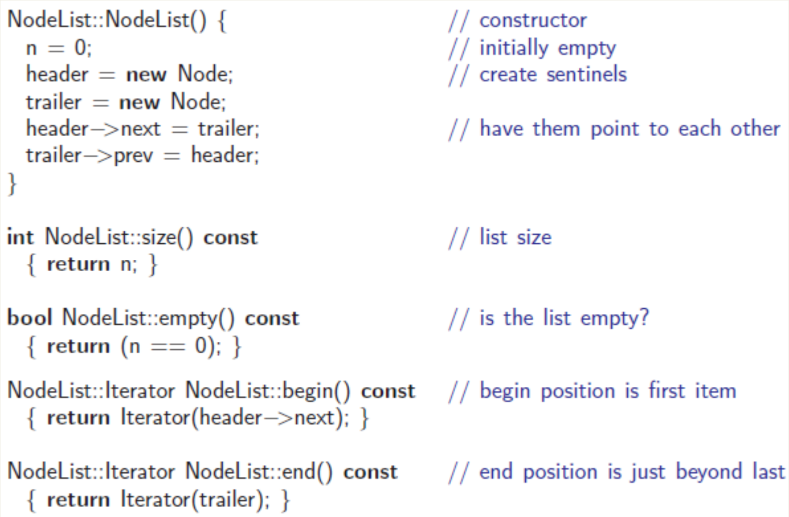
```
class Iterator {                              // an iterator for the list
public:
  Elem& operator*();                          // reference to the element
  bool operator==(const Iterator& p) const; // compare positions
  bool operator!=(const Iterator& p) const;
  Iterator& operator++();                     // move to next position
  Iterator& operator--();                     // move to previous position
  friend class NodeList;                      // give NodeList access
private:
  Node* v;                                    // pointer to the node
  Iterator(Node* u);                          // create from node
};
```



```
NodeList::Iterator::Iterator(Node* u)         // constructor from Node*
  { v = u; }

Elem& NodeList::Iterator::operator*()         // reference to the element
  { return v->elem; }
                                              // compare positions
bool NodeList::Iterator::operator==(const Iterator& p) const
  { return v == p.v; }

bool NodeList::Iterator::operator!=(const Iterator& p) const
  { return v != p.v; }
                                              // move to next position
NodeList::Iterator& NodeList::Iterator::operator++()
  { v = v->next; return *this; }
                                              // move to previous position
NodeList::Iterator& NodeList::Iterator::operator--()
  { v = v->prev; return *this; }
```

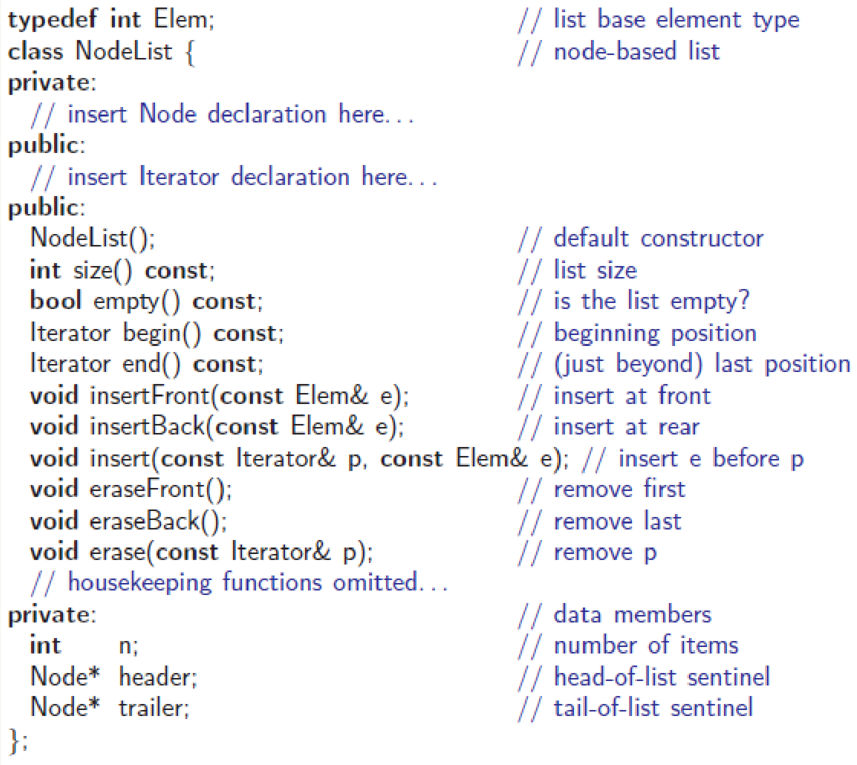


```
typedef int Elem;                             // list base element type
class NodeList {                              // node-based list
private:
  // insert Node declaration here...
public:
  // insert Iterator declaration here...
public:
  NodeList();                                 // default constructor
  int size() const;                           // list size
  bool empty() const;                         // is the list empty?
  Iterator begin() const;                     // beginning position
  Iterator end() const;                       // (just beyond) last position
  void insertFront(const Elem& e);            // insert at front
  void insertBack(const Elem& e);             // insert at rear
  void insert(const Iterator& p, const Elem& e); // insert e before p
  void eraseFront();                          // remove first
  void eraseBack();                           // remove last
  void erase(const Iterator& p);              // remove p
  // housekeeping functions omitted...
private:                                      // data members
  int     n;                                  // number of items
  Node*   header;                             // head-of-list sentinel
  Node*   trailer;                            // tail-of-list sentinel
};
```

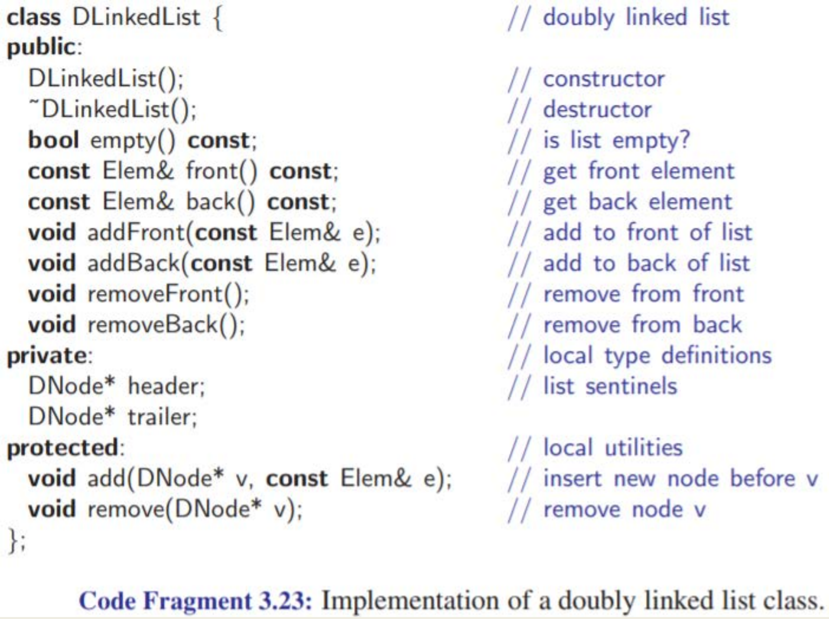


```
NodeList::NodeList() {                        // constructor
  n = 0;                                      // initially empty
  header = new Node;                          // create sentinels
  trailer = new Node;
  header->next = trailer;                     // have them point to each other
  trailer->prev = header;
}

int NodeList::size() const                    // list size
  { return n; }

bool NodeList::empty() const                  // is the list empty?
  { return (n == 0); }

NodeList::Iterator NodeList::begin() const    // begin position is first item
  { return Iterator(header->next); }

NodeList::Iterator NodeList::end() const      // end position is just beyond last
  { return Iterator(trailer); }
```

List (§ 6.2.3) vs. Old DLinkedList (§ 3.3)



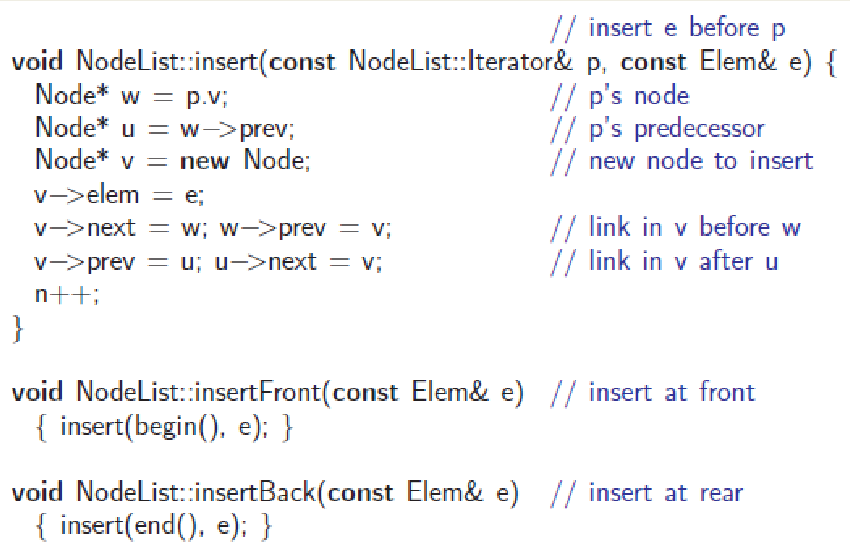
```
typedef int Elem;                              // list base element type
class NodeList {                               // node-based list
private:
  // insert Node declaration here...
public:
  // insert Iterator declaration here...
public:
  NodeList();                                  // default constructor
  int size() const;                            // list size
  bool empty() const;                          // is the list empty?
  Iterator begin() const;                      // beginning position
  Iterator end() const;                        // (just beyond) last position
  void insertFront(const Elem& e);             // insert at front
  void insertBack(const Elem& e);              // insert at rear
  void insert(const Iterator& p, const Elem& e); // insert e before p
  void eraseFront();                           // remove first
  void eraseBack();                            // remove last
  void erase(const Iterator& p);               // remove p
  // housekeeping functions omitted...
private:                                       // data members
  int    n;                                    // number of items
  Node*  header;                               // head-of-list sentinel
  Node*  trailer;                              // tail-of-list sentinel
};
```



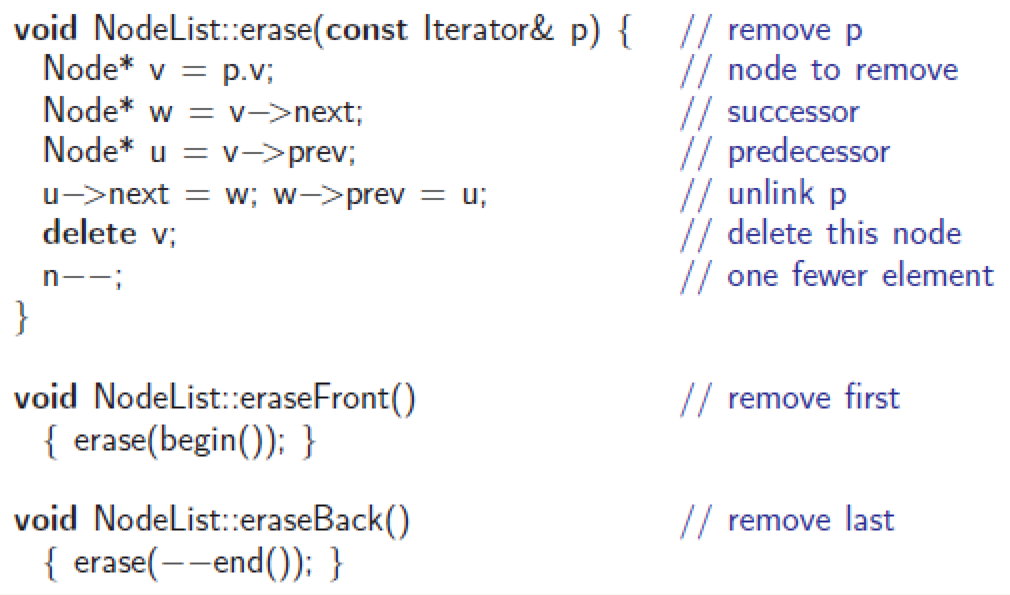
```
class DLinkedList {                            // doubly linked list
public:
  DLinkedList();                               // constructor
  ~DLinkedList();                              // destructor
  bool empty() const;                          // is list empty?
  const Elem& front() const;                   // get front element
  const Elem& back() const;                    // get back element
  void addFront(const Elem& e);                // add to front of list
  void addBack(const Elem& e);                 // add to back of list
  void removeFront();                          // remove from front
  void removeBack();                           // remove from back
private:                                       // local type definitions
  DNode* header;                               // list sentinels
  DNode* trailer;
protected:                                     // local utilities
  void add(DNode* v, const Elem& e);           // insert new node before v
  void remove(DNode* v);                       // remove node v
};
```

Code Fragment 3.23: Implementation of a doubly linked list class.

Doubly Linked List Implementation (§ 6.2.3)

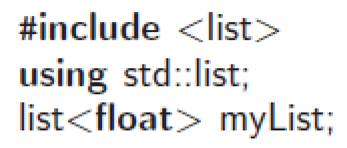


```
                                              // insert e before p
void NodeList::insert(const NodeList::Iterator& p, const Elem& e) {
  Node* w = p.v;                              // p's node
  Node* u = w->prev;                          // p's predecessor
  Node* v = new Node;                         // new node to insert
  v->elem = e;
  v->next = w; w->prev = v;                   // link in v before w
  v->prev = u; u->next = v;                   // link in v after u
  n++;
}

void NodeList::insertFront(const Elem& e)     // insert at front
  { insert(begin(), e); }

void NodeList::insertBack(const Elem& e)      // insert at rear
  { insert(end(), e); }
```



```
void NodeList::erase(const Iterator& p) {     // remove p
  Node* v = p.v;                              // node to remove
  Node* w = v->next;                          // successor
  Node* u = v->prev;                          // predecessor
  u->next = w; w->prev = u;                   // unlink p
  delete v;                                   // delete this node
  n--;                                        // one fewer element
}

void NodeList::eraseFront()                   // remove first
  { erase(begin()); }

void NodeList::eraseBack()                    // remove last
  { erase(--end()); }
```

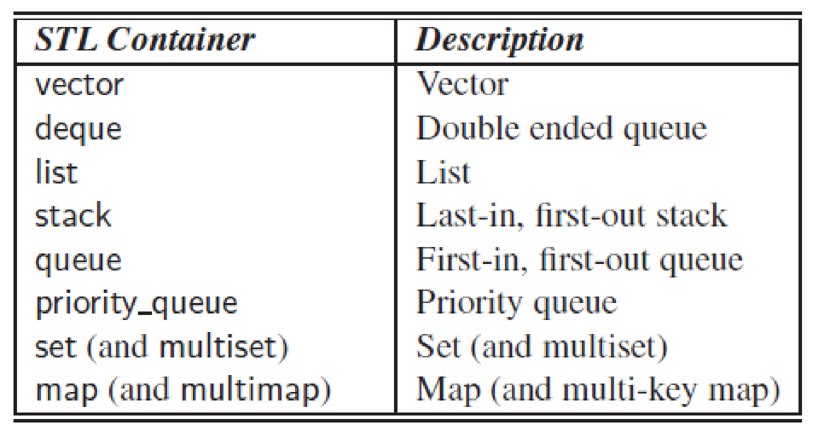
STL Lists (§ 6.2.4)



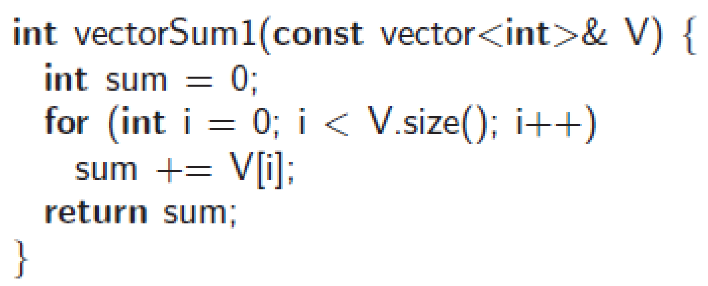
```
#include <list>
using std::list;
list<float> myList;
```

- Implemented using doubly linked list
- Member functions
  - *list(n)*
  - *size()*
  - *empty()*
  - *front()*
  - *back()*
  - *push_front(e)*
  - *push_back(e)*
  - *pop_front()*
  - *pop_back()*
- Note, STL deque manages its elements with a dynamic array and provides random access.

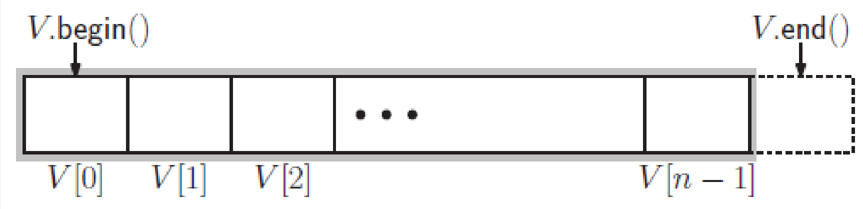
STL Containers and Iterators (§ 6.2.5)

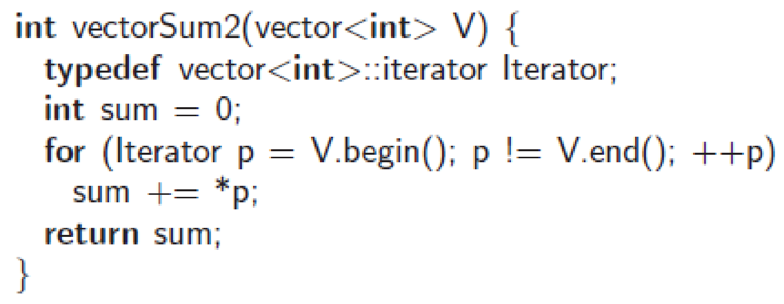


STL Container	*Description*
vector	Vector
deque	Double ended queue
list	List
stack	Last-in, first-out stack
queue	First-in, first-out queue
priority_queue	Priority queue
set (and multiset)	Set (and multiset)
map (and multimap)	Map (and multi-key map)

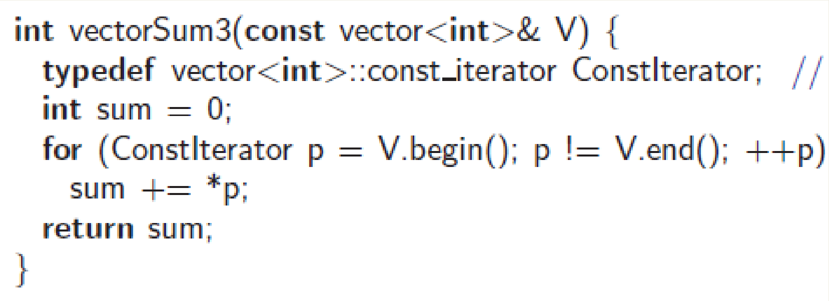


```
int vectorSum1(const vector<int>& V) {
  int sum = 0;
  for (int i = 0; i < V.size(); i++)
    sum += V[i];
  return sum;
}
```





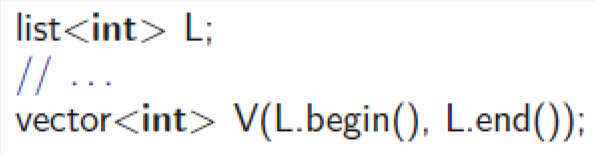
```
int vectorSum2(vector<int> V) {
  typedef vector<int>::iterator Iterator;
  int sum = 0;
  for (Iterator p = V.begin(); p != V.end(); ++p)
    sum += *p;
  return sum;
}
```



```
int vectorSum3(const vector<int>& V) {
  typedef vector<int>::const_iterator ConstIterator;  //
  int sum = 0;
  for (ConstIterator p = V.begin(); p != V.end(); ++p)
    sum += *p;
  return sum;
}
```

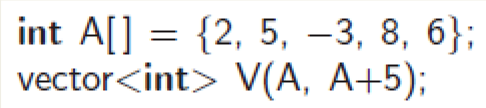
STL Iterator-Based Container Functions

- STL sequence containers (vector, list, deque)
  - *vector(p,q)*
  - *assign(p,q)*
  - *insert(p,e)*
  - *erase(p)*
  - *erase(p,q)*
  - *clear()*



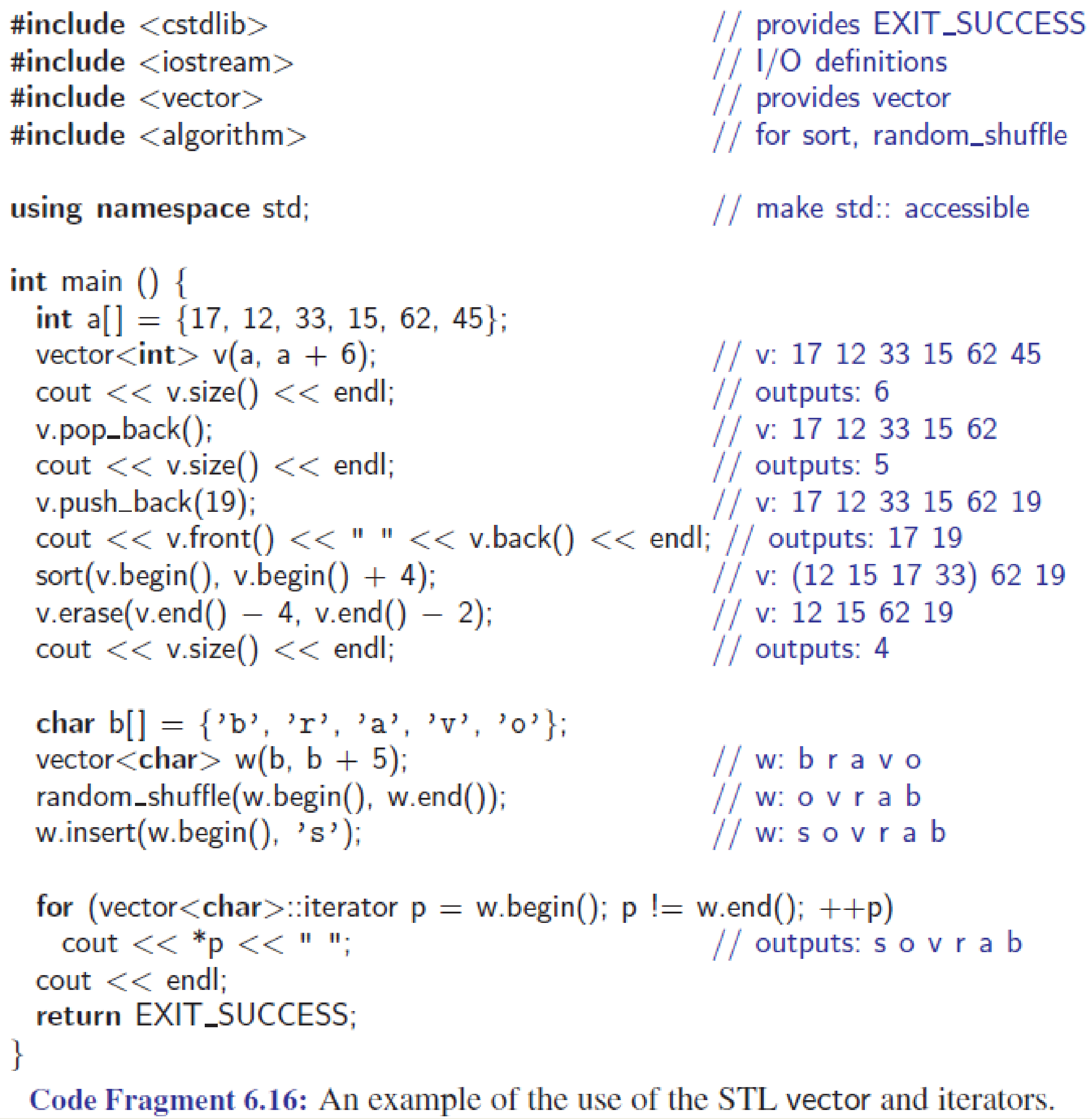
```
list<int> L;
// ...
vector<int> V(L.begin(), L.end());
```

- Pointer arithmetic



```
int A[] = {2, 5, −3, 8, 6};
vector<int> V(A, A+5);
```

- #include<algorithm>
  - *sort(p,q) (no STL list, why?)*
  - *random_suffle(p,q) (no STL list, why?)*
  - *reverse(p,q)*
  - *find(p,q,e)*
  - *min_element(p,q)*
  - *max_element(p,q)*
  - *for_each(p,q,f)*



```
#include <cstdlib>                                    // provides EXIT_SUCCESS
#include <iostream>                                   // I/O definitions
#include <vector>                                     // provides vector
#include <algorithm>                                  // for sort, random_shuffle

using namespace std;                                  // make std:: accessible

int main () {
  int a[] = {17, 12, 33, 15, 62, 45};
  vector<int> v(a, a + 6);                            // v: 17 12 33 15 62 45
  cout << v.size() << endl;                           // outputs: 6
  v.pop_back();                                       // v: 17 12 33 15 62
  cout << v.size() << endl;                           // outputs: 5
  v.push_back(19);                                    // v: 17 12 33 15 62 19
  cout << v.front() << " " << v.back() << endl;       // outputs: 17 19
  sort(v.begin(), v.begin() + 4);                     // v: (12 15 17 33) 62 19
  v.erase(v.end() − 4, v.end() − 2);                  // v: 12 15 62 19
  cout << v.size() << endl;                           // outputs: 4

  char b[] = {'b', 'r', 'a', 'v', 'o'};
  vector<char> w(b, b + 5);                           // w: b r a v o
  random_shuffle(w.begin(), w.end());                 // w: o v r a b
  w.insert(w.begin(), 's');                           // w: s o v r a b

  for (vector<char>::iterator p = w.begin(); p != w.end(); ++p)
    cout << *p << " ";                                // outputs: s o v r a b
  cout << endl;
  return EXIT_SUCCESS;
}
```

Code Fragment 6.16: An example of the use of the STL vector and iterators.