Chapter 9 (part 2): Lists

- Vectors: First-class mechanism for representing lists

Standard Template Library

- What is it?
  - Collection of container types and algorithms supporting basic data structures
- What is a container?
  - A generic list representation allowing programmers to specify which types of elements their particular lists hold
    - Uses the C++ template mechanism
- Have we seen this library before?
  - String class is part of the STL

STL Container Classes

- Sequences
  - deque, list, and vector
    - Vector supports efficient random-access to elements
- Associative
  - map, set
- Adapters
  - priority_queue, queue, and stack

Vector Class Properties

- Provides list representation comparable in efficiency to arrays
- First-class type
- Efficient subscripting is possible
  - Indices are in the range 0 ... size of list – 1
- List size is dynamic
  - Can add items as we need them
- Index checking is possible
  - Through a member function
- Iterators
  - Efficient sequential access

Example

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
  vector<int> A(4, 0); // A: 0 0 0 0
  A.resize(8, 2);      // A: 0 0 0 0 2 2 2 2
  vector<int> B(3, 1); // B: 1 1 1
  for (int i = 0; i < B.size(); ++i) {
      A[i] = B[i] + 2;
  }                    // A: 3 3 3 0 2 2 2 2
  A = B;               // A: 1 1 1
  return 0;
}
```

Some Vector Constructors

- **`vector()`**
  - The default constructor creates a vector of zero length
- **`vector(size_type n, const T &val = T())`**
  - *Explicit* constructor creates a vector of length **`n`** with each element initialized to **`val`**
- **`vector(const T &V)`**
  - The copy constructor creates a vector that is a duplicate of vector **`V`**.
    - Shallow copy!

Construction

- Basic construction

Container name

```
vector<T> List;
```

Base element type

- Example

```
vector<int> A;          // 0 ints
vector<float> B;        // 0 floats
vector<Rational> C;     // 0 Rationals
```

Construction

- Basic construction

Container name

```
vector<T> List(SizeExpression);
```

Base element type

Number of elements to be default constructed

- Example

```
vector<int> A(10);      // 10 ints
vector<float> B(20);    // 20 floats
vector<Rational> C(5);  // 5 Rationals
int n = PromptAndRead();
vector<int> D(n);       // n ints
```

Construction

- Basic construction

Container name

Initial value

```
vector<T> List(SizeExpression, Value);
```

Base element type

Number of elements to be default constructed

- Example

```
vector<int> A(10, 3);      // 10 3s
vector<float> B(20, 0.2);  // 20 0.2s
Rational r(2/3);
vector<Rational> C(5, r);  // 5 2/3s
```

Vector Interface

- **`size_type size() const`**
  - Returns the number of elements in the vector

```
cout << A.size();     // display 3
```

- **`bool empty() const`**
  - Returns true if there are no elements in the vector; otherwise, it returns false

```
if (A.empty()) {
    // ...
```

Vector Interface

- **`vector<T>& operator = (const vector<T> &V)`**
  - The member assignment operator makes its vector representation an exact duplicate of vector V.
    - Shallow copy
  - The modified vector is returned

```
vector<int> A(4, 0); // A: 0 0 0 0
vector<int> B(3, 1); // B: 1 1 1
A = B;               // A: 1 1 1
```

Vector Interface

- **`reference operator [](size_type i)`**
  - Returns a reference to element **`i`** of the vector
    - Lvalue
- **`const_reference operator [](size_type i) const`**
  - Returns a constant reference to element **`i`** of the vector
    - Rvalue

Example

```
vector<int> A(4, 0);                 // A: 0 0 0 0
const vector<int> B(4, 0);           // B: 0 0 0 0

for (int i = 0; i < A.size(); ++i) {
  A[i] = 3;
}                                    // A: 3 3 3 3

for (int i = 0; i < A.size(); ++i) {
  cout << A[i] << endl;               // lvalue
  cout << B[i] << endl;               // rvalue
}
```

Vector Interface

- **`reference at(size_type i)`**
  - If **`i`** is in bounds, returns a reference to element **`i`** of the vector; otherwise, throws an exception
- **`const_reference at(size_type i) const`**
  - If **`i`** is in bounds, returns a constant reference to element **`i`** of the vector; otherwise, throws an exception

Example

```
vector<int> A(4, 0);        // A: 0 0 0 0

for (int i = 0; i <= A.size(); ++i) {
  A[i] = 3;
}  // A: 3 3 3 3 ?? (no run-time error is detected !)
for (int i = 0; i <= A.size(); ++i) {
  A.at(i) = 3;
}// program terminates when i is 4
// (run-time error is detected !)
 //operator[] does not support array-bound checking
  whereas "at" does. Under the same condition, a run-
  time error is detected using "at".
```

Vector Interface

- **`void resize(size_type s, T val = T()`**
  - The number of elements in the vector is now **`s`**.
    - To achieve this size, elements are deleted or added as necessary
      - Deletions if any are performed at the end
      - Additions if any are performed at the end
      - New elements have value **`val`**

```
vector<int> A(4, 0); // A: 0 0 0 0
A.resize(8, 2);      // A: 0 0 0 0 2 2 2 2
A.resize(3,1);       // A: 0 0 0
```

Function Examples

```
void GetList(vector<int> &A) {
  int n = 0;
  while ((n < A.size()) && (cin >> A[n])) {
      ++n;
  }
  A.resize(n);
}
```

```
vector<int> MyList(3);
cout << "Enter numbers: ";
GetList(MyList);
```

Examples

```
void PutList(const vector<int> &A) {
  for (int i = 0; i < A.size(); ++i) {
      cout << A[i] << endl;
  }
}
```

```
cout << "Your numbers: ";
PutList(MyList)
```

Vector Interface

- **`pop_back()`**
  - Removes the last element of the vector
- **`push_back(const T &val)`**
  - Inserts a copy of **`val`** after the last element of the vector

Example

```
void GetValues(vector<int> &A) {
  A.resize(0);
  int Val;
  while (cin >> Val) {
     A.push_back(Val);
  }
}
```

```
vector<int> List;
cout << "Enter numbers: ";
GetValues(List);
```

Overloading >>

```
istream& operator>>(istream& sin, vector<int> &A) {
   A.resize(0);
   int Val;
   while (sin >> Val) {
        A.push_back(Val);
   }
   return sin;
}
```

- // A reference return is more efficient than a standard
- // return. We do not want insertions to go to a copy of cout
- // (temporary stream)
- // rather than to cout. Example:
- // Rational r(1,2); cout<<r<< endl;(ostream&)
- // and cout<< r; cout << endl; (void)

```
vector<int> B;
cout << "Enter numbers: ";
cin >> B;
```

Vector Interface

- **`reference front()`**
  - Returns a reference to the first element of the vector
- **`const_reference front() const`**
  - Returns a constant reference to the first element of the vector

```
vector<int> B(4,1);  // B: 1 1 1 1
int& val = B.front();
val = 7;             // B: 7 1 1 1
```

Vector Interface

- **`reference back()`**
  - Returns a reference to the last element of the vector
- **`const_reference back() const`**
  - Returns a constant reference to the last element of the vector

```
vector<int> C(4,1);  // C: 1 1 1 1
int& val = C.back();
val = 5;             // C: 1 1 1 5
```

Iterators

- Iterator is a pointer to an element
  - Really pointer abstraction
- Mechanism for sequentially accessing the elements in the list
  - Alternative to subscripting
- There is an iterator type for each kind of vector list
- Notes
  - Algorithm component of STL uses iterators
  - Code using iterators rather than subscripting can often be reused by other objects using different container representations

Vector Interface

- **`iterator begin()`**
  - Returns an iterator that points to the first element of the vector
- **`iterator end()`**
  - Returns an iterator that points to immediately *beyond* the last element of the vector

```
vector<int> C(4);  // C: 0 0 0 0
C[0] = 0; C[1] = 1; C[2] = 2; C[3] = 3;
vector<int>::iterator p = C.begin();
vector<int>::iterator q = C.end();
```

Iterators

- To avoid unwieldy syntax programmers typically use typedef statements to create simple iterator type names

```
typedef vector<int>::iterator iterator;
typedef vector<int>::reverse_iterator
 reverse_iterator;
typedef vector<int>::const_reference
 const_reference;
```

```
vector<int> C(4);  // C: 0 0 0 0
iterator p = C.begin();
iterator q = C.end();
```

Iterator Operators

- ***** dereferencing operator
  - Produces a reference to the object to which the iterator **p** points

    ***p**

- **++** point to next element in list
  - Iterator **p** now points to the element that followed the previous element to which p points

    ++p

- **- -** point to previous element in list
  - Iterator **p** now points to the element that preceded the previous element to which p points

    --p

```
typedef vector<int>::iterator iterator;
typedef vector<int>::reverse_iterator reverse_iterator;
vector<int> List(3);
List[0] = 100; List[1] = 101; List[2] = 102;
iterator p = List.begin();
cout << *p;                        // 100
++p;
cout << *p;                        // 101
--p;
cout << *p;                        // 100
reverse_iterator q = List.rbegin();
cout << *q;                        // 102
++q;
cout << *q;                        // 101
--q;
cout << *q;                        // 102
```

Vector Interface

- **`insert(iterator pos, const T &val = T())`**
  - Inserts a copy of **`val`** at position **`pos`** of the vector and returns the position of the copy into the vector
- **`erase(iterator pos)`**
  - Removes the element of the vector at position **`pos`**

SelectionSort Revisited

```
void SelectionSort(vector<int> &A) {
   int n = A.size();
  for (int i = 0; i < n); ++i) {
     int k = i;
     for (int j = i + 1; j < n; ++j) {
        if (A[j] < A[k])
           k = j;
     }
     if (i != k)
        swap(A[k], A[i]);
  }
}
```

QuickSort

- QuickSort
  - Divide the list into sublists such that every element in the left sublist <= to every element in the right sublist
  - Repeat the QuickSort process on the sublists

```
void QuickSort(vector<char> &A, int left, int right) {
  if (left < right) {
      Pivot(A, left, right);
      int k = Partition(A, left, right);
      QuickSort(A, left, k-1);
      QuickSort(A, k+1, right);
  }
}
```

Picking The Pivot Element

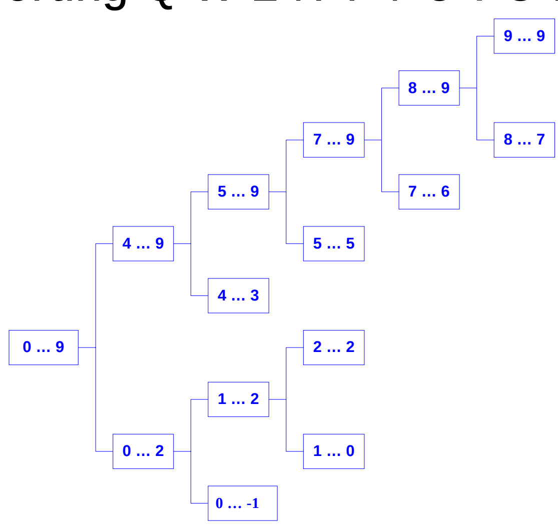
```
void Pivot(vector<char> &A, int left,
  int right) {
  if (A[left] > A[right]) {
     Swap(A[left], A[right]);
  }
}
```

Decomposing Into Sublists

```
int Partition(vector<char> &A, int left, int right) {
  char pivot = A[left];
  int i = left;
  int j = right+1;
  do {
      do ++i; while (A[i] < pivot);
      do --j; while (A[j] > pivot);
      if (i < j) {
            Swap(A[i], A[j]);
      }
  } while (i < j);
  Swap(A[j], A[left]);
  return j;
}
```

Sorting Q W E R T Y U I O P

Q W E R T Y U I O P
I O E P T Y U R W Q
E O I P T Y U R W Q
E O I P T Y U R W Q
E I O P T Y U R W Q
E I O P T Y U R W Q
E I O P T Y U R W Q
E I O P Q Y U R W T
E I O P Q Y U R W T
E I O P Q R T U W Y
E I O P Q R T U W Y
E I O P Q R T U W Y
E I O P Q R T U W Y
E I O P Q R T U W Y
E I O P Q R T U W Y



InsertionSort

```
void InsertionSort(vector<int> &A) {
   for (int i = 1; i < A.size(); ++i) {
      int key = A[i]
      int j = i - 1;
      while ((j > 0) && (A[j] > key)) {
         A[j+1] = A[j]
         j = j - 1
      }
      A[j+1] = key
   }
```

Searching Revisited

- Problem
  - Determine whether a value key is one of the element values in a *sorted* list
- Solution
  - Binary search
    - Repeatedly limit the section of the list that could contain the key value

```
BSearch(const vector<int> &A, int a, int b, int key){
  if (a > b){
      return b+1;
  }
  int m = (a + b)/2
  if (A[m] == key) {
      return m;
  }
  else if (a == b) {
      return -1;
  }
  else if (A[m] < key) {
      return BSearch(A, m+1, b, key);
  }
  else // A[m] > key
      return BSearch(A, a, m-1, key);
  }
```

Run time is proportional to the log of the number of elements

String Class Revisited

```
void GetWords(vector<string> &List) {
  List.resize(0);
  string s;
  while (cin >> s) {
     List.push_back(s);
  }
}
```

Using GetWords()

- Suppose standard input contains

  A list of words to be read.

```
The code is:
    vector<string> A;
    GetWords(A);
```

- Would set **A** in the following manner:

```
A[0]: "A"
A[1]: "list"
A[2]: "of"
A[3]: "words"
A[4]: "to"
A[5]: "be"
A[6]: "read."
```

String Class As Container Class

- A string can be viewed as a container because it holds a sequence of characters
  - Subscript operator is overloaded for string objects
- Suppose **t** is a string object representing **"purple"**
  - Traditional **t** view

```
t: "purple"
```

  - Alternative view

```
t[0]: 'p'
t[1]: 'u'
t[2]: 'r'
t[3]: 'p'
t[4]: 'l'
t[5]: 'e'
```

Example

```
#include <cctype>
using namespace std;
...
string t = "purple";
t[1] = 'e';
t[2] = 'o';
cout << t << endl;          // t: people
for (int i = 0; i < t.size(); ++i) {
  t[i] = toupper(t[i]);
}
cout << t << endl;          // t: PEOPLE
```

Reconsider A

- Where

```
vector<string> A;
```

Is set in the following manner

```
A[0]: "A"
A[1]: "list"
A[2]: "of"
A[3]: "words"
A[4]: "to"
A[5]: "be"
A[6]: "read."
```

Counting o's

- The following counts number of o's within **A**

```
count = 0;
for (int i  = 0; i < A.size(); ++i) {
   for (int j = 0; A[i].size(); ++j) {
      if (A[i][j] == 'o') {
         ++count;
      }
   }
}
```

Size of **A**

Size of **A[i]**

To reference **j**th character of **A[i]** we need double subscripts

Explicit Two-Dimensional List

- Consider definition
  `vector< vector<int> > A;`
- Then
  A is a **vector< vector<int> >**
    - It is a vector of vectors

  A[i] is a **vector<int>**
    - **i** can vary from **0** to **A.size() - 1**

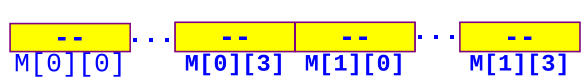
  A[i][j] is a **int**
    - **j** can vary from **0** to **A[i].size() - 1**

Multi-Dimensional Arrays

- Syntax
  `btype mdarray[size_1][size_2] ... [size_k]`
- Where
  - k - dimensional array
  - **`mdarray`**: array identifier
  - **`size_i`**: a positive constant expression
  - **`btype`**: standard type or a previously defined user type and is the base type of the array elements
- Semantics
  - **`mdarray`** is an object whose elements are indexed by a sequence of **`k`** subscripts
  - the **`i`**-th subscript is in the range **`0 ... size_i - 1`**

Memory Layout

- Multidimensional arrays are laid out in row-major order
- Consider
  `int M[2][4];`
- **`M`** is two-dimensional array that consists of 2 subarrays each with 4 elements.
  - 2 rows of 4 elements
- The array is assigned to a contiguous section of memory
  - The first row occupies the first portion
  - The second row occupies the second portion



Identity Matrix Initialization

```
const int MaxSize = 25;
float A[MaxSize][MaxSize];
int nr = PromptAndRead();
int nc = PromptAndRead();
assert((nr <= MaxSize) && (nc <= MaxSize));
for (int r = 0; r < nr; ++r) {
   for (int c = 0; c < nc; ++c) {
      A[r][c] = 0;
   }
   A[r][r] = 1;
}
```

Matrix Addition Solution

Notice only first brackets are empty

```
void MatrixAdd(const float A[][MaxCols],
 const float B[][MaxCols], float C[][MaxCols],
 int m, int n) {
   for (int r = 0; r < m; ++r {
      for (int c = 0; c < n; ++c) {
         C[r][c] = A[r][c] + B[r][c];
      }
   }
}
```