Vectors

Vectors are one of the container class templates defined in the Standard Template Library. To understand vectors, you need to understand template classes.

There are three ways to declare a vector.

Syntax

```
#include <vector>    // must include this file
using namespace std;   // might need this as well

vector<base type> arrayname;            // contains 0 cells
vector<base type> arrayname(size expression);;
vector<base type> arrayname(size expression, value);;
    // contains size-expression cells, each initialized to value
```

The expression can be any expression that evaluates to a number. If the number is not an integer, it is truncated.

Examples

```
vector<int>     grades(5,0);  // vector of 5 ints, all 0
vector<string>  trees(50);    // vector of 50 strings
vector<Point>   hexagon(6);   // vector of 6 Points
    // because Point had a default constructor)

or, more interestingly:

    cout << "Enter the number of sides:";
    cin  >> n;
    vector<Point>  polygon(n);
    vector<double> chordlengths(n*(n+1)/2);

But,

    vector<MyClass>   Object(2);

will be illegal if MyClass does not have a default constructor.

To access an individual element, use the vector name and an index:

```
grades[0]  = 100;
cin >> grade[1];
cout << "The grade is " << grade[1];
for (int i = 0; i < 5; i++)
    cin >> grades[i];
```

To initialize a vector to 0:

```
for (int k = 0; k < 5; k++)
    grades[k] = 0;
```

To compute the average of the values:

```
```
sum = 0.0;
for (int k = 0; k < 5; k++)
    sum += grades[k];
average = sum/5;

**Example**

This simulates rolling a pair of dice with **NSIDES** many sides 20,000 times and counts how many times each possible sum (2,3,4,5,..., 2*NSIDES) occurs.

```cpp
#include <vector>

// use vector to simulate rolling of two dice
const int NSIDES = 4;
int main()
{
    int sum, k;
    Dice d(NSIDES); // Dice defined elsewhere
    vector<int> diceStats(2*NSIDES+1); // room for largest sum
    int rollCount = 20000;

    for (k = 2; k <= 2*NSIDES; k++) // initialize to zero
        diceStats[k] = 0;

    // could have done this at declaration time
    for (k=0; k < rollCount; k++) // simulate all the rolls
    {
        sum = d.Roll() + d.Roll();
        diceStats[sum]++;
    }

    cout << "roll	# of occurrences" << endl;
    for (k=2; k <= 2*NSIDES; k++)
    {
        cout << k << "\t" << diceStats[k] << endl;
    }
    return 0;
}

**vector parameters**

Vectors can be passed as parameters to functions.

```cpp
int Sum(const vector<int> & numbers, int length)
{
    int sum = 0;
    for (int k = 0; k < length; k++)
        sum += numbers[k];
    return sum;
}

void Shuffle(vector<string> & words, int count)
{ 
    RandGen gen;    // for random # generator
    int randWord;
    string temp;
    int k;
    // choose a random word from [k..count-1] for song # k

    for (k=0; k < count - 1; k++)
    {   randWord = gen.RandInt(k,count-1);  // random track
        temp = words[randWord];             // swap entries
        words[randWord] = words[k];
        words[k] = temp;
    }
}

**Collections and Lists Using vectors**

A vector's size is not the same as its capacity. Suppose we have

```cpp
vector<string>   trees(8);
```

and we have filled it with 6 tree names as follows.

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>birch</td>
<td>oak</td>
<td>ebony</td>
<td>cherry</td>
<td>maple</td>
<td>ash</td>
<td>birch</td>
<td>oak</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

The **capacity** is 8 but the **size** is 6. We don't have to keep track of this in our program if we use the methods of the vector class.

The vector class has methods of growing itself and keeping track of how big it is.

```cpp
vector::size()               // returns current size
vector::push_back(value)     // adds another value to a vector
                            // and if it does not have enough
                            // cells it doubles capacity
```

```cpp
vector<double>   prices(1000);   // prices.size() == 1000
vector<int>      scores(20);     // scores.size() == 20
vector<string>   words;          // words.size() == 0;
words.push_back("camel");       // size() == 1, capacity() = 2
words.push_back("horse");      // size() == 2, capacity() = 2
words.push_back("llama");      // size() == 3, capacity() = 4
words.push_back("okapi");      // size() == 4, capacity() = 4
words.push_back("bongo");      // size() == 5, capacity() = 8
```
size() always returns current size, not the number of elements added by push_back. If a vector is initially size 0, and push_back is used exclusively to grow it, size() will return the number of elements pushed onto it.

vector::reserve(size expression)
// allocates an initial capacity but keeps size at 0:

vector<int> votes;
votes.reserve(32000); size() == 0 but capacity == 32000
vector<int> ballots(32000) size() = 32000 and capacity == 32000
for (int i = 0; i < 100; i++){
    cin >> x;
    votes.push_back(x);
} // what is capacity now?

Vector Idioms: Insertion, Deletion, Searching

Typical operations in data processing are:
• insert into a vector (or array)
• delete data from a vector
• search a vector for data

Building an unsorted vector

for (int i = 0; i < 100; i++){
    cin >> x;
    v.push_back(x);
}

or, reading from a file:
vector<double> v;
ifstream fin;
fin.open("inputdata.txt");
double x;
while ( fin >> x ) {
    v.push_back(x);
}

The data are in the order read from the file now.

Deleting from a vector using pop_back()

The pop_back() member function of the vector class deletes the last element of a vector and reduces the size by 1. It does not affect capacity. E.g., assume
vector <double> v(5) contains 8,4,2,10,3
v.pop_back(); => 8 4 2 10
v.pop_back(); => 8 4 2
v.pop_back(); => 8 4

If the vector is unsorted, deletion from position \( \text{pos} \) is easy. We overwrite the item in position \( \text{pos} \) by copying the last element into \( v[\text{pos}] \), then we delete the last element with \( \text{pop\_back()} \):

\[
\text{int lastIndex} = v\text{.size()} - 1;
\text{v[\text{pos}]} = v[\text{lastIndex}];
\text{v.pop\_back();}
\]

**Searching an unsorted vector (linear search)**

To search an unsorted vector it is necessary to look through the entire vector. To look for the cell with the value \( \text{key} \):

\[
\text{int k;}
\text{for (k = 0; k < v.size(); k++){}
\text{if ( v[k] == key )}
\text{break;}
\}
\text{if (k < v.size())}
\text{// not found}
\]

Or, the function:

\[
\text{void LinSearch(const vector<double> & v, double key, int & loc) }
\{ 
\text{int k;}
\text{for (k = 0; k < v.size(); k++){}
\text{if ( v[k] == key )}
\text{loc = k;}
\text{return;}
\}
\text{loc = -1;}
\}
\]

**Sorted vectors**

Vectors can be built in sorted order by inserting data in the right position during creation. This makes later searching faster but makes creation a little slower.

Idea:

\[
\text{while there is more data available}
\text{read the next data item}
\text{let k be the index of the largest element of}
\text{the vector that is smaller than the item}
\text{put this data item into position k+1, shifting}
\text{all larger elements of the vector up one cell}
\]

This is one way to do it. The author does it slightly differently. To be more precise declare
vector<double> sortedNums;
double s;
while there is more data available
    let count = sortedNums.size(); //current size of vector
    cin >> s; //read the next data item s
    sortedNums.push_back(s) //push onto the end of vector
                         //now it has count+1 items
    let k = count;
    while (0 < k && s <= sortedNums[k-1]) {
        sortedNums[k] = sortedNums[k-1];
        k--;
    }

Use the example data
4.5 10 6.3 3.0 1.0

Suppose we have a sorted vector with some large number of items. To delete the item at index n, 0 <= n < size(), we can shift items n+1 to size()-1 down 1 and delete the last:

    for (k = n; k < a.size()-1; k++)
        a[k] = a[k+1];
a.pop_back();

More generally, a function to delete an item from an int vector

    void delete(vector<int> & a, int p)
    {
        int k;
        if ( (p < 0) || (a.size() <= p))
            return;
        for (k = p; k < a.size()-1; k++)
            a[k] = a[k+1];
a.pop_back();
    }

**Searching a sorted vector**

If a vector is sorted we can use more efficient method called binary search.

**Binary Search**

    int bsearch(const vector<string>& list, const string& key)
    // precondition: list.size() == # elements in list
    // postcondition: returns index of key in list, -1 if key not found
    {
        int low = 0; // leftmost possible entry
        int high = list.size()-1; // rightmost possible entry
        int mid;
        while (low <= high)
```c
{  
    mid = (low + high)/2;
    if (list[mid] == key) // found key, exit search
    {      return mid;
    }
    else if (list[mid] < key) // key in upper half
    {      low = mid + 1;
    }
    else // key in lower half
    {      high = mid - 1;
    }
    return -1; // not in list
}
```

**Example**

Search for each of: "ash" "kapok" "elm" in

```
ash  birch  cherry  dogwood  ebony  imbuya  kapok  maple
```