Vectors and Arrays

Arrays

- Available regardless of which platform you use
- Easy to understand
- Low-level and fast
- Easy to initialize

A **homogeneous aggregation** is a collection of variables in which all members of the collection are the same type. C and C++ provide a homogeneous aggregation called an **array**.

Motivation: Arrays are used for manipulating collections of data.

An array is conceptually a linear collection of elements, indexed by subscripts:

45	89	77	95	60
0	1	2	3	4

Elements can be any type (called the base type). For example, they can be ints, strings, structs, or classes, **provided that the base type has a default (i.e. parameterless) constructor**.

birch	oak	ebony	cherry	maple
0	1	2	3	4

Subscripts or **indices** are the "names" for the individual elements of the vector or array. If the above array was declared as

```
string    trees[5];
```

then `trees[2]` has the value "`ebony`" and `trees[4]` has the value "`maple`".

Defining an array

```
basetype arrayname[size expr];
basetype arrayname[size expr] = { list with <= sizeexpr vals };
basetype arrayname[ ]         = { list with any number of  values };
```

Examples

```
#define  MAXWORDS  20000

const    int SIZE = 100;
double   numbers[SIZE];
string   words[MAXWORDS];
string   days[7]    = {"Sun","Mon","Tues","Wed","Thurs","Fri","Sat"};
int      powers[ ]   = {0,1,2,4,9,16,25,36,49,64,81,100};
int     counts[100] = {0};
```

```
int     counts[100] = {1}; // <-- does not work
```

- Arrays can also be initialized using loops.
- Cannot assign one array to another.
- Array parameters are always passed by reference -- they are called ***array parameters*** because you do not need to put the reference operator & to the left of the name. It is enough to put brackets next to the parameter name.

Example

```
int coins[ ] = {1,5,10,25,50};
void display ( ostream & output , int list[], int size)
{
     for (int i = 0; i < size; i++) {
          output << list[i];
          output << (i < size-1?",":"\n");
     }
}
```

Implementation note: Array names are really pointers to the first cell of the array.

Vectors

Vectors are one of the container class templates defined in the Standard Template Library. 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.

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

vector parameters

Vectors can be passed as parameters to functions.

```
int Sum(const vector<int> & numbers, int length)
{
     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

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

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

birch	oak	ebony	cherry	maple	ash		
0	1	2	3	4	5	6	7

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.

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

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 `pos` is easy. We overwrite the item in position `pos` by copying the last element into `v[pos]`, then we delete the last element with `pop_back()`:

```
int lastIndex = v.size() – 1;
v[pos] = v[lastIndex];
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 key:

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

Or, the function:

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

```
while there is more data available
    read the next data item
    let k be the index of the largest element of
       the vector that is smaller than the item
    put this data item into position k+1, shifting
       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;                        // middle of current range
    while (low <= high)
    {
        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