C++ Vector Class

Limitations of Arrays

- **The size of arrays must be a known constant prior to compile time.**
  - – Inefficient use of space
    - e.g. A program needs enough storage to maintain 1000 real numbers, but on average, the user only enters 10 real elements. So, 900 * 8bytes = 7200bytes wasted on average.
  - – Arrays cannot be resized in your program
    - What if a user needs 2030 elements? Then the above program is useless!

Limitations of Arrays (2)

- **The size of arrays must be maintained persistently throughout the entire program.**
  - It is not always possible to find the size of any array using the `sizeof()` function. For example, think about an array of strings.
  - A function allowing an array as its argument must also allow an array size argument.

```
int getMin(const int v[], const int V_SIZE)
```

Limitations of Arrays (3)

- In a function, if an array parameter is not supposed to be written over (just read), the `const` keyword should be used to declare the array parameter to ensure safety from writes.

- Likewise, the array size parameter should also be constant (since arrays cannot be resized)

```
int getMin(const int v[], const int V_SIZE)
```

Vector Class

- The C++ vector class is an alternative to using regular arrays, and helps us avoid array limitations.

- To use vectors, we must

```
#include <vector>
```

- To declare a vector, the C++ syntax is

```
vector<dataType> varName;
```

Declaring Vectors (1)

- Something you can do with arrays:
  - Declare a vector, `v1[]`, with 10 integers:

    ```
    vector<int> v1(10);
    ```

  - Declare a vector, `v2[]`, with 25 characters:

    ```
    vector<char> v2(25);
    ```

Declaring Vectors (2)

- Some things you cannot do with arrays:
  - Ask for a positive integer, n, then declare a vector, `v2[]`, with n strings:

    ```
    cout << "How many strings need to be stored?";
    cin >> n;
    vector<strings> v2(n);
    ```

  - Declare a vector, v4, with 5 doubles, and initialize all values to -1.0 on the fly:

    ```
    vector<double> v4(5, -1.0);
    ```

Vector Methods (1)

`size()`	Returns the size of the calling vector.	`cout << “The size of my vector, vals, is: ” << vals.size();`

`insert(pos, elem)`	Inserts element `elem` into calling vector at position `pos`.	`// creates a vector, v10`
		`// of 0 strings.`
		`vector<string> v10;`
		
		`// inserts “David” into`
		`// first position`
		`v10.insert(0, “David”);`

`push_back(elem)`	Inserts element `elem` at the end of the calling vector.	`//v3 is empty`
		`vector<int> v3;`
		`v3.push_back(54);`
		`v3.push_back(19);`
		`//v3 now has 54 and 19`

Vector Methods (2)

`empty()`	Returns true if the calling vector is empty, and false otherwise.	`if ( v1.empty() )`
		`{`
		`   // v1 is empty`
		`}`

`erase(pos)`	Removes the element at position `pos`.	`//delete 1st element`
		`v2.erase(0);`
		
		`//delete last element`
		`V2.erase(v2.size()-1)`

`erase(b, e)`	Remove all elements within the specified range [`b`, `e`].	`//delete 2nd to 5th elements`
		`v2.erase(1, 4);`

`clear()`	Removes all elements from the calling vector.	`//delete all elements`
		`v2.clear();`

Example

```
int main()
{
   //a vector of 5 doubles all initialized to -1.2
   vector<double> v(5, -1.2);
   double sum = 0.0;

   for (int i=0; i<v.size(); i++)
   {
        //notice vector elements are accessed the same
        //way that array elements are accessed
        sum += v[i];
   }

   //should output -6 here
   cout << sum << endl;

   return 0;
}
```

Example as Function Parameter

- Here, we’re illustrating `findMax()` with vectors.
- Notice the convenience of the size() method. The actual size of the vector no longer has to be known a-priori and passed.

```
int findMax(vector<int> values)
{
  int max = values[0];
  for (int i=1; i<values.size(); i++)
  {
    if (max < values[i])
    {
        max = values[i];
    }
  }
  return max;
}
```

Vectors Are Not Always Passed by Reference!

- The following function ***will not*** alter vector the contents in vector A[] as before.

```
int main()
{
    vector<double> A(2, 0.0);
    resetVector(A);    //nothing happens to A
}

//resets all values of vector B to 0.0
void resetVector(vector<double> B)
{
    for (int i=0; i<B.size(); i++)
    {
          B[i] = 0.0;
    }
}
```

So, how do we ensure that A[] is reset zeroes? (1)

- **Solution one: explicitly state that the vector should be passed by reference.**

```
int main()
{
    vector<double> A(2, 0.0);
    resetVector(A);     //A is now reset
}

//resets all values of vector B to 0.0 by reference
void resetVector(vector<double> &B)
{
    for (int i=0; i<B.size(); i++)
    {
          B[i] = 0.0;
    }
}
```

So, how do we ensure that A[] is reset zeroes? (2)

- **Solution two: Pass the vector back to the caller (remember, this could not be done with arrays)**

```
int main()
{
     vector<double> A(2, 0.0);
     A = resetVector(A);  //see below
}

//returns a vector of doubles
vector<double> resetVector(vector<double> B)
{
     for (int i=0; i<B.size(); i++)
     {
            B[i] = 0.0;
     }
     return B;
}
```

Multidimensional Vectors (1)

- A 2D matrix is essentially a vector of vectors. The following declares a 2 x 5 matrix of doubles:

```
vector< vector<double> > v1(2, 5);
```

- The space between the last two `> >` is required! (Why do you think this is so?)

Multidimensional Vectors (2)

- Obtaining row size is easy. Since 2D vectors are a vector of vectors, the row size must be:

  `v1.size()`

- Obtaining column size takes one more step. It is the size of any element of `v1`. Since all elements of `v1` are vectors themselves, the column size is:

  `v1[0].size()`

Multidimensional Vector Traversal

- Traversing a multidimensional vector is not unlike traversing a multidimensional array.
- In fact, the `[][]` operator still applies. The following prints out all elements of v2:

```
//for each row
for (int i=0; i<v2.size(); i++)
{
    //for each column
    for (int j=0; j<v2[0].size(); j++)
    {
            cout << v2[i][j] << endl;
    }
}
```

So, Why Use Arrays at All?!

- Overhead
  - The nice convenient functions that vectors offer over arrays introduce overhead with respect to speed and space.

- When should arrays be used over vectors?
  - When the dataset does not need to be contracted or expanded.
  - When speed is an issue.