Basic Vectors Facilities

Arrays are a programming tool that provide a mechanism to store a group of values under a single name. The values can be any available data type (e.g., **`int`**, **`double`**, **`string`**, etc.). In C++, we talk about vectors, rather than arrays.

Vectors are declared with the following syntax:

```
vector<type> variable_name (number_of_elements);
```

The number of elements is optional. You could declare it like this:

```
vector<type> variable_name;
```

And that would declare an *empty* **`vector`** — a **`vector`** that contains zero elements.

The argument **`type`** in angle-brackets indicates the data type of the elements of the vector; **`variable_name`** is the name that we assign to the vector, and the optional **`number_of_elements`** may be provided to indicate how many elements the vector will initially contain.

Below are several examples of **`vector`** declarations:

```
vector<int> values (5);       // Declares a vector of 5 integers
vector<double> grades (20);   // Declares a vector of 20 doubles
vector<string> names;         // Declares a vector of strings,
                              // initially empty (contains 0 strings)
```

When using vectors in our programs, we must provide the appropriate **`#include`** directive at the top, since vectors are a Standard Library facility, and not a built-in part of the core language:

```
#include <vector>
```

After a vector has been declared specifying a certain number of elements, we can refer to individual elements in the vector using square brackets to provide a *subscript* or *index*, as shown below:

```
grades[5]
```

When using a vector or array followed by square brackets with a subscript, the resulting expression refers to one individual element of the vector or array, as opposed to the group of values, and roughly speaking, you can use that expression as you would use a variable of the corresponding data type. In the above example, the data type of the expression **`grades[5]`** is **`double`**, so you can use it as you would use a variable of type double — you can assign a value to it (a numeric value, with or without decimals), or you can retrieve the value, use it for arithmetic operations, etc.

The above extends to other data types as well; if we have a **vector** of **string**s called **names**, the expression **names[0]** is a **string**, referring to the first element in the vector **names**. We can do anything with this expression that we would do with a **string** variable. For instance, the expression **names[0].length()** gives us the length of this string.

An important condition for the index or subscript is that it must indicate a valid element in the vector. Elements in a vector are "numbered" starting with element 0. This means that valid subscript values are numbers between **0** and **size-1**, where **size** is the number of elements of the vector. For the example above of **grades**, valid subscripts are between **0** and **19**.

The following fragment shows an example of a program that asks the user for marks for a group of 20 students and stores them in a vector.

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<double> student_marks(20);

    for (vector<double>::size_type i = 0; i < 20; i++)
    {
        cout << "Enter marks for student #" << i+1
             << ": " << flush;
        cin >> student_marks[i];
    }
    // ... Do some stuff with the values

    return 0;
}
```

The first statement declares a vector called **student_marks** with capacity to hold 20 values of type **double**. These values can be accessed individually as **student_marks[0]** to **student_marks[19]**. The **for** loop has the counter **i** go from **0** to **19**, allowing access to each individual element in a sequential manner, starting at **0** and going through each value from **0** to **19**, inclusively.

Notice the data type for the subscript, **vector<double>::size_type**. As with strings, class **vector<type>** provides a **size_type** to represent positions and sizes. It is always recommended that you use this data type when dealing with vectors.

for loops usually go hand in hand with the use of vectors or arrays, as they provide a convenient way to access every element, one at a time, using the loop control variable as the subscript. This does *not* mean that we must use **for** loops whenever we require access to the elements of a vector — it only means that quite often, a **for** loop provides a convenient approach and we choose it as the mechanism to access the elements.

Resizing Vectors

Vectors have one important advantage with respect to C-style arrays: vectors can be resized during the execution of the program to accommodate any extra elements as needed, or even to "shrink" the vector.

In the example from the previous fragment above, if we don't know ahead of time (i.e., at the time we are writing the program) that there are 20 students, we could obtain that information at run-time (e.g., prompt the user for the number of students) and resize the vector accordingly, as shown below (though we notice that the example is somewhat silly, in that we could have waited until having the value of **num_students** and then declare the **vector** initializing it with that size):

```
vector<double> student_marks;
    // no size specified: vector contains
    // no elements

int num_students;
cout << "Number of students: " << flush;
cin >> num_students;

student_marks.resize (num_students);

for (vector<double>::size_type i = 0; i < num_students; i++)
{
    cout << "Enter marks for student #" << i+1
         << ": " << flush;
    cin >> student_marks[i];
}
```

Notice that the valid subscripts for a vector with **num_students** elements are **0** to **num_students-1**. For that reason, the **for**loop starts at **0** and goes while **i** is less than **num_elements**.

It is always a better idea to control **for** loops using the **size** method of **vector**. That way, we make sure that we loop only through the right subscript values, and we avoid the risk of accidentally exceeding the limits of the vector:

```
for (vector<double>::size_type i = 0; i < student_marks.size(); i++)
```

The difference in this case seems insignificant, and it almost sounds unnecessary to use the **size** method; but again, it's always a good idea to stick to good programming practices that may be very convenient in larger or more complex programs.

In some situations, we can not determine the number of elements before reading them. That is, we may have to read numbers to then determine when to stop reading them (an example would be, keep reading values until you read a negative value). In such situations, the trick of resizing the **vector** is not an option (at least not the way it is used in the example above).

Vectors provide a convenient way of handling this type of situation. We can use the **push_back** method to append one element at the end of the array. The operation includes resizing to one more element to accommodate for the extra element, and storing the given value at the end of the array.

The example below shows the use of **push_back** to accept numbers from the user and store them in a vector, until the user indicates that there are no more numbers.

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<double> student_marks;
    double mark;
    char answer;

    cout << "Enter marks (y/n)? " << flush;
    cin >> answer;

    while (answer == 'y' || answer == 'Y')
    {
        cout << "Enter value: " << flush;
        cin >> mark;

        student_marks.push_back (mark);

        cout << "More students (y/n)? " << flush;
        cin >> answer;
    }

    return 0;
}
```

Inserting and Removing Elements

Methods **push_back** and **pop_back** insert and remove (respectively) elements at the end of the vector. For situations where we need to insert or remove at an arbitrary position, we have methods **remove** and **remove**. We notice that these methods are inefficient with a **vector** (they take linear time, since all the remaining elements from the given position to the end have to be shifted). However, for situations where we must support these operations, class **vector** does provide the facilities to do so.

Methods **insert** and **remove** use *iterators* as parameters to indicate the position at which we want to insert or remove the element(s). However, a “quick and dirty” solution is available, given the nature of vector iterators (they support operations essentially identical to pointer arithmetic). The code sample below illustrates these features:

```
#include <vector>
using namespace std;

int main()
{
    vector<double> values(10);
        // Create vector with 10 elements (initialized to 0.0)

    values.insert (values.begin() + 5, 1.4142);
        // Insert sqr root of 2 at position 5 (before the element that
was at position 5)

    values.remove (values.begin() + 3);
        // Remove element at position 3
```

In the sample above, the expressions **`values.begin() + n`** work similarly to the idea of getting a pointer to element n of an array — we get a pointer to the first element of the array, then add an integer offset to obtain a pointer pointing n elements after the beginning of the array. The expression **`values.begin()`** returns an iterator pointing to the first element of the vector, and adding an integer value n results in an iterator pointing n positions after the first element, providing the required parameter for **`insert`** and **`remove`** methods.