Built in One-Dimensional Arrays

A picture is worth a thousand words. This is an array:

[0]	[1]	[2]	[3]	[4]	[5]	[6]
9.3	-63.12	16.0	0.45	-4.1	54.3	47.2

An array is a data structure that can hold many variables of the same type. Each of the variables in an array is called an *element* or an *entry*. The example above shows a 7-element array of `doubles`. The entries of an array can be of any defined data type. The entries are numbered starting at zero, so in the example, 9.3 is the beginning or zero element, and 47.2 is the sixth element.

Here is code to setup the above array:

```
double numbers[7] = {9.3, -63.12, 16.0, 0.45, -4.1, 54.3, 47.2};
```

In general, normal or built-in arrays in C++ are defined as follows.

```
double stuff[7];                    // A 7-element array of doubles
int numbers[10];                    // A 10-element array of
integers
```

Each element of an `array` can be referred to individually as if it were not part of an `array` at all. Given the declarations above, `numbers[0]` refers to the first element of the `array` `numbers`, `numbers[1]` refers to the second, and `numbers[9]` refers to the last. Thus we can write:

```
cin >> numbers[0];            // Gets input for the first entry
cout << numbers[0];           // Displays the value of the first
entry
```

The real advantage of `arrays` is that they allow programs to manipulate many numbers (or letters, or whatever) without having to use a lot of variables. Consider the following code:

```
string names[10];
for (int i=0; i<10; i++)
  cin >> names[i];
```

This commonly used fragment gets 10 `strings` from the user in only two lines of code. This idea—using a loop to apply a process to the elements of an `array`, one by one—is one of the most commonly used of all programming techniques.

Built in arrays may be passed as parameters in functions. They are by default referenced, so you are modifying the original. They have no default value, and cannot be resized, so remember to initialize, and use them carefully. An example of arrays in functions follows.

```
#include <iostream>

using namespace std;

void getData(string names[10]);

int main()
{
  string names[10];
  getData(names);
  .
  .
  return 0;
}

void getData(string names[10])
{
  cout<<"Please enter 10 names:";
  for (int i=0; i<10; i++)
    cin >> names[i];
}
```

One-Dimensional Arrays Implemented as `vector` Objects

Normal arrays in C++ allow an unfortunately high opportunity for disastrous misuse, especially to beginning programmers. However, arrays can be safely declared if we use the `vector` class, which is defined in the include file `vector` . Some sample arrays could be defined as follows:

```
#include <vector>
…
vector<double> stuff(7);            // A 7-element array of doubles
vector<int> numbers(10);            // A 10-element array of integers
```

Each element of a `vector` can be referred to individually as if it were not part of a `vector` at all. Given the declarations above, `numbers[0]` refers to the first element of the `vector` `numbers`, `numbers[1]` refers to the second, and `numbers[9]` refers to the last. Thus we can write:

```
cin >> numbers[0];            // Gets input for the first entry
cout << numbers[0];           // Displays the value of the first entry
```

It is also worth mentioning the `vector` class object can be declared and each element of the vector filled with a given value. For example:

```
vector<double> stuff(7, 10.3);    // All elements in stuff
contain 10.3
vector<int> numbers(10, 5);       // All elements in numbers
contain 5
```

The real advantage of `vectors` is that they allow programs to manipulate many numbers (or letters, or whatever) without having to use a lot of variables. Consider the following code:

```
vector <string> names(10);
for (int i=0; i<10; i++)
  cin >> names[i];
```

This commonly used fragment gets 10 `strings` from the user in only two lines of code. This idea—using a loop to apply a process to the elements of a `vector`, one by one—is one of the most commonly used of all programming techniques.

Vectors may be passed as parameters in functions. They should always be passed by reference to ensure efficient use of memory. Because you always pass by reference you are always capable of modifying the original. The solution to this problem is the const keyword. By adding the const keyword in front of the vector definition in the parameter list, the memory efficiency is maintained, while the vector is un-modifiable. An example of vectors in functions follows.

```
#include <iostream>
#include <vector>

using namespace std;

void getData(vector<string> &names);
void display(const vector<string> &names);

int main()
{
  vector<string> names(10," ");
  getData(names);
  display(names);
  .
  .
  return 0;
}

void getData(vector<string> &names)
{
  cout<<"Please enter 10 names:";
  for (int i=0; i<10; i++)
    cin >> names[i];
}

void display(const vector<string> &names)
{
  cout<<"Here are your 10 names:";
  for (int i=0; i<10; i++)
    cout<< names[i];
}
```

Selected `vector` member functions

Member	Use
`=`	Assigns one `vector` to another `vector` and automatically resizes the first to the same size as the one being assigned
`[n]`	Returns the $(n+1)^{th}$ element of the `vector`. (Indexes start at zero.)
`clear()`	Removes all of the elements from the `vector`. (Empties it to size 0)
`empty()`	Returns true if the `vector` has no elements, false otherwise
`resize(k)`	Adds or deletes elements from the `vector` to give `k` elements. Added elements will be initialized at the default value if one was used to construct the `vector`.
`size()`	Returns the number of elements in the `vector`
`swap(v)`	swap the contents of this vector with vector v