C++ Arrays

An array is a set of continuous memory locations all holding the same kind of data.

It is often the case when writing programs that one needs a large number of storage locations. Ordinary variables have to be declared individually, and are not well suited for this purpose.

For example, if we wish to have 5 float variables to hold test scores, one might make the following declaration:

```
float Score1, Score2, Score3, Score4, Score5;
```

In a program, input of scores might look something like this:

```
cout << “Enter Score 1: “;
cin >> Score1;
cout << “Enter Score 2: “;
cin >> Score2;
cout << “Enter Score 3: “;
cin >> Score3;
cout << “Enter Score 4: “;
cin >> Score4;
cout << “Enter Score 5: “;
cin >> Score5;
```

Imagine what would happen if we wanted to read 100 scores!

Arrays make the declaration of the 5 float variables an easy task. The following declaration will do the job.

```
float Score[5];
```

This declaration makes available 5 storage locations in which to put floats. The individual **array members** or cells of the array defined above are denoted by:

```
Score[0],  Score[1],  Score[2],  Score[3],  Score[4]
```

The values 0, 1,…4, used to access the value of an array cell are called **subscripts**. In C and C++, subscripts always begin at 0.

The type of data that is actually in the array is called the **base type**. In the array above, the base type is float, but it can be any type, i.e. a programmer can set up an array to hold char, int, long, etc. All members of a single array must be the same data type.

One of the features of arrays that make them so powerful is that variables can be used to specify an array location. This allows loops to process arrays. For example, to initialize the locations of the above array to 0, the loop below could be used

```
for ( int k = 1; k <= 4; ++k)
  Score[k] = 0.0;
```

The following picture shows what the array above has after the loop is done.

	0	1	2	3	4
Score	0.0	0.0	0.0	0.0	0.0

Declaration of Arrays

An array declaration is of the form “`DataType ArrayName[SIZE];`”. For example, an array of 100 characters would be declared as:

```
    char Ch[100];
```

Note that although the “upper limit” 100 is used in the declaration of the array, it is not the last available subscript, which is 99.

Arrays can also be filled with information when declared. For example, to declare and fill an int array with the first 8 primes, one could use the declaration

```
  int A[8] = {2, 3, 5, 7, 11, 13, 17, 19};
```

If not enough values are included in the initialization list, the remainder of the array values are set to 0. For example,

```
  int B[10] = {1};    //initialize B[0] to 1 and B[1] .. B[9] to 0
```

A programmer does not have to count the number of initializing elements. The declaration below also works:

```
  int A[] = {2, 3, 5, 7, 11, 13, 17, 19};
```

Arrays of char

Array of characters are especially important, because text processing is so common. In C and C++, arrays of char that have a null character (ASCII 0) to mark the end of the array are often called **strings**. There is some extra built-in language support for these strings. For example, consider the declaration below:

```
  char Str[] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’};
```

C++ allows an easier way to declare this array, namely:

```
  char Str[] = “Hello”;
```

As a bonus, you get a null character added to the end of the array. This allows you to make a single cout statement to display the array, for example:

```
  cout << Str;   // output: “Hello”,
```

Only null terminated arrays of char can be output in this way. Other arrays require a loop to output the contents of the array one at a time.

One is also able to read in strings from user input, for example:

```
  cin >> Str;    // user types in “Good”
  cout << Str;   // output: “Good”
```

Note that when reading strings , “cin >> “ follows a rule similar to the one it uses to read numbers or characters. Leading whitespaces are skipped, then a contiguous group of characters are read and stored in array, and input stops when trailing whitespace is encountered, and null terminator is placed at end of array. Fortunately the *iostream* library provides a way (`cin.getline()`) to read an entire phrase, blanks and all.

A Common Programming Error

One of the most common C++ programming errors is referencing an array "member" that is not really in the array, i.e. using too large a subscript. In most programming languages, such as FORTRAN, BASIC or Java, such an error causes abnormal program termination. However, in C++ and C there is no check to make sure that array subscripts are within proper limits. Using an out of bounds subscript often causes mysterious run-time problems or causes program to crash. The program below illustrates common program behavior when out of bounds subscripts are used.

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

void main()
{
  int A[8] = {2, 3, 5, 7, 11, 13, 17, 19};
  int B = 1234;

  cout << "A[8] = " << A[8] << endl; //possible output: A[8] = 1234
  A[8] = 0;

  cout << "B = " << B << endl;       //possible output: B = 0
}
```

Debugging Tip: If when running one of your programs, you find that the value of a variable changes unexpectedly, check to make sure that you are not writing information to an "array member" that is really beyond the end of the array.

An Array is a Constant Pointer

Many programming languages view an array as a symbol for the entire bank of memory locations. In these languages, an assignment statement using arrays, such as A1 = A2, transfers all of the members of A2 to A1. However, in C and C++, an array is viewed as a constant pointer that always points to or holds the address of the first member of the array. This means that an array name is consider a constant and cannot appear on the left-hand side of an assignment operator! The program segment below shows a common array syntax error.

```
char A[5];
A = "Hello";
```

This is a syntax error, since you are trying to change the value of the constant A by storing in it the address of the string "Hello".

The fact that an array is an address has implications when passing arrays to functions. Consider the short program below.

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

void StoreX( char A[] )
{
  int N = 0;

  while ( A[N] != 0 )
    A[N++] = ‘x’;
}

void main()
{
  char Str[] = “Hello”;

  StoreX(Str);

  cout << “Str = “ << Str << endl;   //output: Str = xxxxx
}
```

When the call “`StoreX(Str)`” is made, the address of the array is passed to the caller, function StoreX. This allows StoreX to get at the array members and change them. In fact, the heading of StoreX can also be declared as follows:

```
void StoreX( char *A )
```

This declaration style emphasizes the fact that A is a pointer (address).

Because passing an address is associated with passing a parameter by reference, some programmers say that arrays are passed by reference in C and C++. Technically, however, this is not true. When an array is passed to a function, the address of the array is passed, but if the function changes this address, it is restored upon return to main(), just like other value parameters. Nonetheless, it is probably wise to think of array as the bank of storage locations that are passed by reference. Another, perhaps better way of thinking, is to view arrays as a special case: they are passed by *address*.