Chapter 8: Arrays

Notes

- The address of the first subscript of an array can be used as if it *were* an array. Thus array[0] and (&array)[0] are identical.
- To set a pointer to an array you can use the array name or get the address of the first subscript.
- Pointer arithmetic can utilize an integer-based number as long as it is not used for direct assignment (p = x is bad, but p += x is not).

Introduction

Using a structure you can group several members under a single name, even group member structures. But to store a sequence of values you still have to give each one a unique name. Thus if you wanted to store three '`int`' values, you might create three variables:

```
int num1, num2, num3;
```

If you then wanted to modify all of these in the same way you'd have to do it individually. For example if we wanted to add one to each one, you might write:

```
num1++;
num2++;
num3++;
```

This might be fine for a couple variables, but it gets to be an inconvenience as you need more values. A different way to approach this is using *arrays*, which is one or more values of the same data type grouped under a single name. Each value part of an array is known as a *subscript*, similar to the *members* of structures, and is accessed via an numerical *index* value rather than a member name.

Array Declaration

One way to create an array is a normal variable declaration followed by a number enclosed in square brackets:

```
data_type name[number];
```

This declares an array of '*data_type*' values, under '*name*', with '*number*' amount of subscripts. The following then would create an array of 'int' values under the name 'num', with three (3) subscripts:

```
int num[3];
```

That is, three 'int' variables are created as part of a single glob. With a structure, each member occupies a piece of a single glob of memory. The single glob is the name of the variable and each member has a unique name. Arrays are similar in that each subscript occupies a piece of a single glob of memory. The single glob is the name of the array and each subscript has a unique index.

The size of an array in declaration is a constant amount for the number of subscripts. Since it is a constant amount it cannot be specified using *any* variable value. An expression can be used only if it involves *constant* or *literal* values. The following are valid array declarations:

```
const int cx = 10;
int num[cx];
int num[10];
int num[5 + 5];
```

While the following are *invalid* array declarations:

```
int x = 10;
int y = 5;
int num[x];
int num[5 + x];
```

An expression devoid of variable parts is valid in an array declaration because it can be calculated without knowing anything outside of itself. For example: 'int num[x];' cannot be calculated unless we can get the value of 'x'; whereas 'int num[10];' we are using a literal value that is *known.* Thus, the compiler will calculate these constant expressions and replace them with a single number rather than keep them a set of instructions to be executed.

You may be wondering how constants fit into this. Constants are named values that *never change*. This allows the compiler to know across the entire scope of the constant what value it is and will always be.

Subscripts

A less ambiguous example of an array would be a list of ten favorite numbers:

```
int favorite[10];
```

Each subscript is a single variable with an 'int' value; each is a part of 'favorite' and accessed via a numerical index. The index is a zero-based number. That is, the first index is zero (0) rather than one (1). The second index is two (2) and so on. The tenth and last index is nine (9). When you declare an array you specify the *amount* of subscripts (or indices[1]) *not* the last valid index. The 'favorite' array has *ten* (10) subscripts, zero (0) to nine (9). Thus the last index is one minus the amount of subscripts.

You can refer to each subscript (piece of an array) by naming the array and following it with a *subscript index*, or simply *index*, enclosed in square brackets ([ ]); known as the *index operator*. Usually "subscript *X*" is said, where "*X*" is a number, rather than "subscript at index *X*"; the latter is valid, but takes inconveniently longer to say ☺.

In our array of favorite numbers, you would access the first favorite number value, "subscript 0", by specifying: '`favorite[0]`'. This is seen as "subscript 0 of the array named '`favorite`'" or more simply "subscript 0 of '`favorite`' array". An array subscript, then, can be used anywhere you would use a normal variable:

```
favorite[0] = 5;
cout << "First favorite number = " << favorite[0] << endl;
int x = favorite[0] + 5;
cout << "Enter YOUR favorite number: ";
cin >> favorite[0];
cout << "You entered " << favorite[0] << endl;
```

Any integer-resulting expression can be used as a subscript. Thus you can use an integer variable, integer constant, integer literal, function returning an integer, a complex expression resulting in an integer, etc:

```
#include <iostream.h>

int GetZero(void) { return 0; }
int GetFour(void) { return 4; }

int main()
{
    int favorite[10];
    int x = 0;
    const int i = 0;
    favorite[x] = 5;
    favorite[i] = 5;
    favorite[x + i + 2 - 2] = 0;
    favorite[GetZero()] = 0;
    favorite[GetFour() - 4] = 0;
    return 0;
}
```

Looping Subscripts

[1] The terms 'indexes' and 'indices' are both correct and synonymous with each other.

One of the most common occurrences of arrays is in concert with loops. The loop will increment a variable from zero up to the last index in an array. That variable is used as the index for the array. For example, the following would set all of the subscripts in '`favorite`' to zero (0):

```
int favorite[10];
int i = 0;
while (i < 10)
{
    favorite[i] = 0;
    i++;
}
```

The name '`i`' is typically used to mean "index". In the above, the array '`favorite`' is created with ten subscripts. A '`while`' loop is then used to increment '`i`' while it is less than '10' (the number of subscripts in 'favorite'). In each loop iteration the subscript specified by '`i`' is set to zero. A more orthodox approach is the '`for`' loop for this purpose:

```
int favorite[10];
int i;
for (i = 0; i < 10; i++)
{
    favorite[i] = 0;
}
```

Blarg.

Initializing Arrays

Arrays can be initialized like other variables when they are declared. This initialization looks exactly like the initialization for a structure. The array is declared in the same way except it is followed up by an equal sign and a block with a list of values separated by commas.

```
data_type name[number] = { value0 [, …] };
```

The following would initialize the array 'favorite' to ten zeroes:

```
int favorite[10] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
```

The number of values used in the initialization can be equal to or *less* than the number of subscripts in the array. If you use fewer values then only some of the subscripts are initialized. For example, the following initializes the first *five* subscripts of 'favorite' to zero, the rest will not be initialized and will contain garbage data:

```
int favorite[10] = { 0, 0, 0, 0, 0 };
```

The values used in initialization can be any expression that equates to a value of the same data type as the array. For the array ‘favorite’, one could use an integer variable, integer constant, integer literal, function returning an integer, or a complex expression resulting in an integer:

```
#include <iostream.h>

int GetZero(void) { return 0; }
int GetFour(void) { return 4; }

int main()
{
    int x = 0;
    const int i = 0;
    int favorite[10] = { 0, x, i, GetZero(),
                         GetFour() - 4 };
    return 0;
}
```

This initializes the first five subscripts of the array to zero (0) like before, but uses five different approaches: a literal ‘0’, a variable ‘`x`’, a constant ‘`i`’, a function return value ‘`GetZero()`’, and a complex expression resulting in an ‘int’: ‘`GetFour() - 4`’.

Declaring Arrays without Size

It is possible, and used quite often, to declare an array without specifying a size. An array of this kind must be initialized immediately. The size of the array will depend on the amount of initializers used. Thus, if you initialize five (5) subscripts, it will be as if you declared an array to hold five (5) subscripts. The following are identical:

```
int array[5] = { 0, 1, 2, 3, 4 };
int array[] = { 0, 1, 2, 3, 4 };
```

The latter method is done without specifying a size. This is useful when creating arrays in a program that contain a table of known values that may change.

Copying an Array

An array is a list of values. To copy the values of one array to another, you must do it a single subscript at a time. The following does not work:

```
int array1[5] = { 0, 1, 2, 3, 4 };
int array2[5] = array1;
```

Since the array is multiple values, you must copy each of those values. Assignment between arrays does not work[2]. Thus, to copy all of ‘array1’ to ‘array2’ you might do this:

```
int i = 0;
for (i = 0; i < 5; i++)
    array2[i] = array1[i];
```

Blarg.

Arrays in Memory

If each value is bananas stored in a crate, then an array is a *string* of crates. That is, each crate in an array is part of a group. It can be used independently, but it is also part of a whole. You may have heard the term “string” apply to a variable that stores text: e.g. one or more characters. You will learn more about character strings and how they pertain to arrays in the next chapter.

Although each subscript is part of a whole, using an array without an index represents a memory address equal only to the first subscript:

```
int array[5];
cout << array << “ / “ &array[0] << endl;
```

The above would print out the memory address of ‘`array`’ as well as the address of its first subscript. The two are the same. Can you guess why? It is because of how the array is stored in memory. It is a string of values and therefore a string of bytes. The first value is at the top of the string and subscripts of the group itself are accessed by an offset from that address. That is why the first subscript is at index zero: it is accessing a value at offset zero (0) (i.e. none) from the memory address of the array.

Therefore an array is somewhat like a pointer with less functionality. It represents a memory address. In fact, you can even dereference the array to get access to the first member:

```
#include <iostream.h>

int main()
{
    int array[5];
    cout << array << “ / “ << &array[0] << endl;
    *array = 7;
    cout << array[0] << endl;
    return 0;
}
```

[2] “Smart” arrays can use assignment operators if they are designed for them. They are covered much later, be content with copying by subscript for now.

The output on my computer was this:

```
0x0065FDD0 / 0x0065FDD0
7
```

On yours the memory address printed will probably be different, but still be duplicated. The second line is seven (7) because using '`*array`' is identical to using '`array[0]`'. Both access the value at offset zero (0) in the array.

You cannot, however, reassign an array to something else. The following will *not work*:

```
int array1[5];
int array2[5];
array1 = array2;
```

There are other operations as well that work on pointers, but not on arrays.

Pointers to Arrays

Pointers have a very unique relationship with arrays. They can represent them without any special operators. That is if you declare a pointer, and set it to an array, it can be used in the same way you used the array variable. A pointer to an array must be declared only with the same data type. Thus if you wanted to create a pointer to an array of type '`int`':

```
int *pointer;
```

Since an array identifier by itself represents the address of the first subscript; assigning a pointer to an array is easy. Simply use the array name by itself when assigning the array to a pointer:

```
int array[5];
int *p = array;
```

The above creates a pointer 'p' to 'array'. This pointer can be used exactly how you would use 'array' by itself:

```
#include <iostream.h>

int main()
{
    int array[5];
    int *p = array;
    p[2] = 7;
    cout << array[2] << endl;
    return 0;
}
```

The output of this program is simply seven (7). Be careful when using the index operator (`[ ]`) on pointers; make sure the index you are using exists. You can set a pointer to the memory address of a non-array variable and still use the index operator. In that case, the only index that is valid is zero (0) because it is not a list of values, but only one. Remember that dereferencing a pointer is identical to using the index operator with zero. So '`*p`' is the *same* as '`p[0]`'.

Pointer Arithmetic

There are even more advantages to having a pointer to an array. You can do special math operations on the pointer, known as *pointer arithmetic*. Because a pointer's value is a numerical memory address, it makes sense that you can do math operations with it. However, the effect of these is different than that of normal variables:

```
int array[5];
int *p = array;
p++;
```

What is the memory address of '`p`' after the above? Obviously it would no longer be the same as '`array`'. The memory address of '`p`' will *not* be incremented by one byte, but will be incremented by the size of '`int`'.

Any addition and subtraction operations on a pointer with integer numbers, will affect that pointer's address in units equal to the size of the pointer's data type. Thus, a '`char`' pointer will be affected in terms of bytes, but a '`short`' pointer will be affect in terms of words (two bytes). For example:

```
short array[5];
short *p = array;
p += 2;
p--;
*p = 7;
```

The result of the above is the subscript of 'array' at index one (1) is set to seven (7). The first pointer arithmetic is 'p += 2'. This does not add *two* (2) to the pointer's memory address; it adds four (4). The storage unit of a 'short' is two (2) bytes and therefore is altered in terms of two (2) bytes, not one (1). The next arithmetic is 'p--' which will decrement the memory address of 'p' by two (2) or the size of a 'short' storage unit. This sounds rather complicated; just think of it this way: a 'short' pointer represents a 'short' value and therefore must be altered two (2) bytes at a time.

When performing arithmetic between two pointers, the result is absolute and the data type of the pointers are irrelevant. That is, if you subtract one pointer from another, the result is in terms of raw bytes, not the data type represented by the pointer. There are some specialized occasions when you will want to use pointers together in an operation.

However, those occasions are a bit more complex and we will visit them in the next chapter.

Mathematical operations other than addition and subtraction are *not allowed* with pointers. There is no reason for them. The only arithmetic you need with pointers is that which allows you to move them up or down throughout memory; multiplication and division do not fit this usage.

Using simple addition with pointers you can simulate the index operator. All you have to do is add an offset amount to the pointer, dereference it, and *voila*! For example:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int array[5];
06      int *p = array;
07      *(p + 2) = 7;
08      cout << array[2] << endl;
09      return 0;
10  }
```

This is identical to a previous example, save for line seven (7). We add two (2) to ‘p’ and dereference the resulting memory address which is the same as ‘array[2]’. Thus, using the index operator is the same as using a pointer with basic addition arithmetic. In fact, this arithmetic can be done on arrays so long as it does not modify the original address.

References to Arrays

References to arrays are illegal, period.

Constant Arrays

Arrays can be a list of constants rather than variables. Declaring a constant array is the same as declaring a variable array, with the exception of the keyword ‘const’ that precedes the declaration. The following declares an array of ten (10) ‘int’ constants:

```
const int array[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

You must initialize *every* member in a constants array, because they can never be changed once they are set (e.g. they are *constant*). A constant array acts the same as a variable array except you cannot *change* any of the subscript values. Since you must initialize every member, you may find it easier to omit the size in the declaration:

```
const int array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

This equates to the same thing. For every initializer a new subscript is added. The above declaration is the same as the one before it; because it has ten (10) initializers and thus an array with ten (10) subscripts is created.

Array Function Parameters

There are three ways to specify a function parameter that accepts a variable array. The first is to specify an array with a size:

```
void foo(int array[10]);
```

The above function appears to accept an '`int`' array with ten (10) subscripts. This is only partially correct. The size of an array when used as a function parameter is *ignored*. Thus this way is identical to the second way which is specifying an array without a size:

```
void foo(int array[]);
```

This new above function accepts any '`int`' array regardless of the amount of subscripts. Thus you could pass it an array with ten (10) subscripts or three hundred (300); it does not distinguish between them. The reason is that when you pass an array to a function, you are only passing the *memory address of the array*, not the array itself. If you were to modify 'array' inside '`foo`', it would change the array you originally passed in. This brings me to the third method for specifying an array parameter, and that is by pointer:

```
void foo(int *array);
```

The last prototype of '`foo`' above will accept an '`int`' array *or* a pointer. A dangerous fact in addition is that you can pass a pointer to a function accepting an array type as well! That is, all of the '`foo`' functions prototyped above could be used with '`int`' pointers rather than '`int`' arrays. Remember that an array is simply the starting memory address of a single glob of memory.

Function prototypes do not require the *names* of parameters, only information about them. As far as array parameters go you do not have to specify the size of the array or its name, but you must include the brackets:

```
void foo(int []);
```

The above function accepts an 'int' array. Remember to name the parameter when you actually define the function.

Passing Arrays to Functions

When you want to pass an array to a function you use only its name. If you specify a subscript you will be passing that subscript to the function and not the entire array. So passing an array to a function is like passing any other variable to a function: using *only* its name. Take a look at the following example:

```
#include <iostream.h>

void printarray1(int array[], int size);
void printarray2(int *array, int size);

int main()
{
    int array[3] = { 0, 1, 2 };
    printarray1(array, 3);
    printarray2(array, 3);
    return 0;
}

void printarray1(int array[], int size)
{
    int i;
    cout << “Printing using Array Parameter” << endl;
    for (i = 0; i < size; i++)
        cout << “#” << i << “ = “ << array[i] << endl;
}

void printarray2(int *p, int size)
{
    int i;
    cout << “Printing using Pointer Parameter” << endl;
    for (i = 0; i < size; i++)
        cout << “#” << i << “ = “ << p[i] << endl;
}
```

The output of the above program is:

```
Printing using Array Parameter
#0 = 0
#1 = 1
#2 = 2
Printing using Pointer Parameter
#0 = 0
#1 = 1
#2 = 2
```

Both functions act identically, but one uses a pointer parameter and one uses an actual array parameter. An array parameter is created an initialized as if it was a pointer, but then has the same syntactical restrictions as an array. That is an array parameter acts exactly like it was an actual array of its own and follows those rules, but underneath it is actually a pointer. Why? Passing something to a function parameter is like declaring and initializing a variable. Calling ‘printarray2’ then is like creating ‘p’ and initializing it to the address of our array:

```
int *p = array;
```

That makes sense, but if we take this literally for all parameters then calling 'printarray1' should look like this:

```
int array[] = array;
```

Yes, these two arrays have the same name, but since they exist in different scopes it is okay. Beyond that, this is not valid. You cannot create an array and assign it the value of another array; it is illegal as previously explained. What actually happens is that the parameter 'array' is seen as an 'int' pointer rather than an 'int' array, which is legal:

```
int *array = array;
```

However, when you actually use an array parameter it will *act* as if it was an array. That is, you can use only non-modifying pointer arithmetic on it and it cannot be used in an assignment operation without specifying a subscript.

Author's Preference: Unless I have a good reason to, I usually specify functions accepting arrays through pointers. It is more dangerous but there are fewer restrictions, and you don't run into as many problems with constant arrays (as you will soon see).

SizeOf Array

I used a 'size' parameter in my example above to specify the number of subscripts in the array passed in. Using the 'sizeof' keyword with a primitive data type yields the amount of bytes it uses in memory. But this is only sometimes true for arrays and never when used with an array parameter.

An array parameter is actually a pointer, as mentioned earlier. I can prove it easily by using the 'sizeof' keyword to determine the byte size of it. This keyword only knows things within a limited context. Although we are passing in array, the keyword itself only sees the parameter which will be used to reference the array. The parameter itself will have a low size (4 bytes on a 32-bit system like Win95/98/ME or 2 bytes on a 16-bit system like RealMode MS-DOS) while the array itself consumes much more memory. You can see this if you add the following at the end of 'printarray1':

```
cout << "sizeof(array) = " << sizeof(array) << endl;
```

And add this at the end of 'printarray2':

```
cout << "sizeof(p) = " << sizeof(p) << endl;
```

And lastly add this under the *call* to 'printarray2' (above 'return 0;'):

```
cout << "sizeof(array) = " << sizeof(array) << endl;
```

The output now looks like this on my machine (and probably yours as well unless you're in the future or living in the past):

```
Printing using Array Parameter
#0 = 0
#1 = 1
#2 = 2
sizeof(array) = 4
Printing using Pointer Parameter
#0 = 0
#1 = 1
#2 = 2
sizeof(p) = 4
sizeof(array) = 12
```

As you can see the 'array' parameter in 'printarray1' only uses four (4) bytes which is conspicuously the same size as the pointer 'p' in 'printarray2'. But the end of the program reveals that 'sizeof' outputs the entire byte size when used with an actual array variable. Why is that? Remember how an array parameter is actually a pointer under the guise of an array? Well, an actual array refers to a single block of memory that the compiler knows about (since you declared it) and can act, partially, like a pointer.