Arrays and Pointers

How Arrays Are Stored in Memory

The elements of a one-dimensional array are effectively a series of individual variables of the array data type, stored one after the other in the computer's memory. Like all other variables, each element has an address. The *address of an array* is the address of its first element, which is the address of the first byte of memory occupied by the array.

A picture can help illustrate this. Let's say we declare an array of six integers like this:

```
int scores[6] = {85, 79, 100, 92, 68, 46};
```

Let's assume that the first element of the array `scores` has the address 1000. This means that `&scores[0]` would be 1000. Since an `int` occupies 4 bytes, the address of the second element of the array, `&scores[1]` would be 1004, the address of the third element of the array, `&scores[2]` would be 1008, and so on.

	[0]	[1]	[2]	[3]	[4]	[5]
scores	85	79	100	92	68	46
	1000	1004	1008	1012	1016	1020

What about a two-dimensional array declared like this?

```
int numbers[2][4] = {{1, 3, 5, 7}, {2, 4, 6, 8}};
```

We usually visualize a two-dimensional array as a table of rows and columns:

	column			
row	[0]	[1]	[2]	[3]
[0]	1	3	5	7
[1]	2	4	6	8
	numbers			

But the reality is that a two-dimensional array in C++ is stored in one dimension, like this:

row	[0]				[1]			
column	[0]	[1]	[2]	[3]	[0]	[1]	[2]	[3]
numbers	1	3	5	7	2	4	6	8
	1000	1004	1008	1012	1016	1020	1024	1028

All of the elements of row 0 come first, followed by the elements of row 1, then row 2, etc. This is known as row-major order.

Array to Pointer Conversion

Array names in C++ have a special property. Most of the time when you use an array name like `scores` in an expression, the compiler implicitly generates a pointer to the first element of the array, just as if the programmer had written `&scores[0]`. If the type of the array is T, the data type of the resultant pointer will be "pointer-to-T". For example, `scores` is an array of `int`, so the pointer generated by the compiler will be data type `int*` ("pointer to an `int`").

There are three exceptions where this implicit conversion does **not** happen:

1. **When the array name is the operand of a `sizeof` operator.** For example, the expression `sizeof(scores)` will resolve to 24 (the size of the array in bytes), not 8 or 4 (the size of a pointer to the first element of the array).
2. **When the array name is the operand of the `&` operator.** For example, the expression `&scores` will resolve to the address of the array, not the address of the first element of the array. The address of the array and the address of the first element of the array are in fact the same number, but they are different data types. The data type of the address of the first element of the array is `int*` ("pointer to an int"), while the data type of the address of the array itself is `int (*)[6]` ("pointer to an array of 6 ints").
3. **When the array is a string literal initializer for a character array.** For example:

```
char text[] = "Happy New Year";
```

Here are some examples of when an array name will be converted into a pointer to the first element of the array:

- Using an array name with the subscript operator. This converts the array name to a pointer to the first element and then performs the equivalent of pointer arithmetic followed by dereferencing.
- Using an unsubscripted array name with the dereference operator, `*`. This converts the array name to a pointer to the first element and then dereferences the pointer, giving you the value of the first element of the array.
- Using an unsubscripted array name as a function argument. This converts the array name to a pointer to the first element, passing a copy of the address of the first element of the array to the function. As a result, arrays are passed by address by default.
- Using an unsubscripted array name on the right side of an assignment statement. This converts the array name to a pointer to the first element which can be used in pointer arithmetic or assigned to a pointer variable of the appropriate type.
- Using an unsubscripted array name as the operand of a relational operator. (This is why you can't compare C strings using the relational operators - what you end up comparing is the addresses of the first elements of the two arrays, not the contents of the two arrays).
- Returning the name of an array data member. This actually returns a pointer to the first element.
- Using an unsubscripted array name with the stream insertion operator `<<`. This converts the array name to a pointer to the first element and prints the address of that element, unless the array is an array of `char`. In that case, the `<<` operator assumes that the array is a C string, so it loops through the characters of the array and prints them until it encounters a null character.
- Using an unsubscripted name of an array of `char` with the stream extraction `>>` operator. The array name is converted to a pointer to the first element of the array and passed to the overloaded `operator>>()` function, which uses it to read characters into the original array in the calling routine.

There are some cases where the implicit pointer conversion takes place but produces a syntax error. For example:

- Using an unsubscripted array name on the left side of an assignment statement. The array name is converted into a pointer to the first element of the array, but that pointer is not an lvalue (it's a prvalue) and therefore cannot be used on the left side of an assignment statement.
- Using the unsubscripted array name with the `++` or `--` operators. The array name is converted into a pointer to the first element of the array, but that pointer is not an lvalue and therefore cannot be used with the increment and decrement operators, which recquire their operand to be an lvalue).
- Using an unsubscripted name of an array of any type other than `char` with the stream extraction `>>` operator. This does not work because there is no version of the overloaded `operator>>()` function defined for those data types.

Here's a short program example illustrating the syntax errors described above:

```
// test.cpp
#include <iostream>

using std::cin;

int main()
{
    int array1[6] = {1, 2, 3, 4, 5, 6};
    int array2[6];

    array1++;

    array2 = array1;

    cin >> array2;

    return 0;
}
```

If you try to build this program, you'll get error messages similar to the following:

```
test.cpp: In function 'int main()':
test.cpp:11:11: error: lvalue required as increment operand
    array1++;
          ^~
test.cpp:13:14: error: invalid array assignment
    array2 = array1;
             ^~~~~~
test.cpp:15:9: error: no match for 'operator>>' (operand types are 'std::istream {aka std::basic_istre
    cin >> array2;
    ~~~~^~~~~~~~~
[200+ lines of other mostly useless error output produced by this error]
```

(When C++ fails to find a match for a call to an overloaded function, it will list all of the possible "candidate" functions and then tell you why each one does not match. For the function `operator>>()`, there are 25(!) possible candidates, so that results in a **lot** of error output.)

Passing Arrays to Functions

As mentioned above, passing an unsubscripted array name as an argument to a function converts the array name into to a pointer to the first element of the array. That means the array is passed by address, **which means the function it is passed to can change the values stored in the original array.**

For example:

```
#include <iostream>

using std::cout;
using std::endl;

//prototype: note the notation for an array of int: int[]
void increment_array(int[]);

int main()
{
    int i;

    int numbers[4] = {1, 2, 3, 4};

    increment_array(numbers);

    for (i = 0; i < 4; i++)
        cout << numbers[i] << " ";       // Prints 2, 3, 4, 5
    cout << endl;

    return 0;
}

void increment_array(int a[])
{
    int i;

    for (i = 0; i < 4; i++)
        a[i]++;      // this alters values in numbers in main()
}
```

Since the unsubscripted array name `numbers` is converted into a pointer to the first element of the array when passed as an argument to the `increment_array()` function, we can even use the notation for "pointer to an `int`" as the data type of the argument instead of the notation for "array of int":

```
#include <iostream>

using std::cout;
using std::endl;

//prototype: note the notation for a pointer to an int: int*
void increment_array(int*);

int main()
{
    int i;

    int numbers[4] = {1, 2, 3, 4};

    increment_array(numbers);

    for (i = 0; i < 4; i++)
        cout << numbers[i] << " ";       // Prints 2, 3, 4, 5
    cout << endl;

    return 0;
}

void increment_array(int* a)
{
    int i;

    for (i = 0; i < 4; i++)
        a[i]++;      // this alters values in numbers in main()
}
```

As you can see, we can still use the subscript operator with the pointer `a`, even though it has been defined as a "pointer to an `int`" instead of as an "array of `int`". In fact, it is legal syntax in C++ to use the subscript operator with *any* pointer, regardless of whether or not it actually points to an array element. However, if the pointer doesn't in fact point to an array element, it's usually a bad idea.

Passing Array Elements to Functions

If we pass a single **element** of an array as an argument to a function, it is typically passed by value (i.e. a copy is passed) by default since an element of an array is normally a **simple** data item, not an array. Imagine a new function, using `numbers` as declared above, which has the following prototype

```
void fn(int);
```

and is called like this:

```
fn(numbers[i]);
```

Here, `fn()` *cannot* alter `numbers[i]` no matter what it does, since `numbers[i]` is a simple integer (or whatever type `numbers` was declared as).

Of course, if we really *needed* the function to alter `numbers[i]`, we could change the prototype and function definition so that the argument was passed by reference instead of by value.

Pointer Arithmetic

Incrementing a pointer variable makes it point at the next memory address for its data type. The actual number of bytes added to the address stored in the pointer variable depends on the number of bytes an instance of the particular data type occupies in memory. On our Unix system, for example:

- incrementing a `char*` adds 1
- incrementing an `int*` adds 4
- incrementing a `double*` adds 8

But the important idea is that the pointer now points to a new address, exactly where the next occurrence of that data type would be. This is exactly what you want when you write code that loops through an array - just increment a pointer and you are pointing at the next element. You don't even have to know how big the data type is - C++ knows. In fact you could take code that works for our Unix system (with 4-byte integers) that uses a pointer variable to loop through an array and recompile it on a system that uses 2-byte or 8-byte integers and the code would still work just fine.

All arithmetic involving pointers is scaled based on the data type that the pointers involved point to. For example, if we declare the following array

```
int scores[6] = {85, 79, 100, 92, 68, 46};
```

which can be visualized like this

	[0]	[1]	[2]	[3]	[4]	[5]
scores	85	79	100	92	68	46
	1000	1004	1008	1012	1016	1020

we can write code like the following:

```
int* ptr1 = &scores[2]
ptr1 += 3;                      // ptr1 now points to element 5 of the array

int* ptr2 = &scores[4];
ptr2 -= 2;                      // ptr2 now points to element 2 of the array

int distance = ptr1 - ptr2;     // distance = 3 (three elements between ptr1 and ptr2)
```

Given a pointer to a memory location, you can add to or subtract from it and make it point to a different place in memory.

- `(scores + 0)` points to the integer 85
- `(scores + 1)` points to the integer 79
- `(scores + 2)` points to the integer 100
- etc.

Notice that these expressions are always of the form `<ptr-to-something> + int`. The "`ptr-to-something`" part of the expression is sometimes called the *base address* and the integer added to it is called the *offset*.

So the expression:

```
*(scores + 2) = 90;
```

changes the third array element from 100 to 90. That's the exact same result as using the subscript notation:

```
scores[2] = 90;
```

As far as the C++ compiler is concerned, **array and pointer subscripting doesn't really exist**. Expressions that the programmmer writes using subscript notation are automatically converted to the "base address plus offset" notation. To the compiler, all four of the following expressions produce the same result:

`ar[i]`	`*(ar + i)`	`*(i + ar)`	`i[ar]`

That last expression is kind of silly, but it works, and it should help make it clear that the compiler simply converts expressions that subscript arrays or pointers into the "base address plus offset" notation.

The following table shows the values and data types of various expressions using the array name `scores`:

Expression	Value	Data Type	Expression	Value	Data Type
`scores[0]`	85	`int`	`*(scores+0)`	85	`int`
`scores[1]`	79	`int`	`*(scores+1)`	79	`int`
`scores[2]`	100	`int`	`*(scores+2)`	100	`int`
`scores[3]`	92	`int`	`*(scores+3)`	92	`int`
`scores[4]`	68	`int`	`*(scores+4)`	68	`int`
`scores[5]`	46	`int`	`*(scores+5)`	46	`int`
`&scores[0]`	1000	`int*`	`(scores+0)`	1000	`int*`
`&scores[1]`	1004	`int*`	`(scores+1)`	1004	`int*`
`&scores[2]`	1008	`int*`	`(scores+2)`	1008	`int*`
`&scores[3]`	1012	`int*`	`(scores+3)`	1012	`int*`
`&scores[4]`	1016	`int*`	`(scores+4)`	1016	`int*`
`&scores[5]`	1020	`int*`	`(scores+5)`	1020	`int*`
`*scores`	85	`int`	`*scores+1`	86	`int`
`scores`	1000	`int*`	`&scores`	1000	`int (*)[6]`

Processing Arrays Using Pointer Arithmetic

Pointer arithmetic is frequently used when looping through arrays, especially C strings. For example, suppose we want to write a function to change a C string to all uppercase.

```
char s[80] = "some stuff";
string_to_upper(s);     //the calling statement; passes address of s[0]
```

We could easily write this function using the familiar subscript notation for accessing array elements:

Version 1: Subscript Notation

```
void string_to_upper(char str[])
{
    int i;

    for (i = 0; str[i] != '\0'; i++)
        str[i] = toupper(str[i]);
}
```

However, since the unsubscripted name of an array used as a function argument is converted by the compiler into a pointer to the first element of the array, we could just as easily treat the incoming argument as a pointer to a `char` and write the function using pointer notation:

Version 2: "Base Address Plus Offset" Pointer Notation

```
void string_to_upper(char* str)
{
    int i;

    for (i = 0; *(str + i) != '\0'; i++)
        *(str + i) = toupper(*(str + i));
}
```

The above code is practically identical to the subscript notation version, which should not be surprising if you keep in mind that `str[i]` and `*(str + i)` produce exactly the same effect.

There's another way to write the function using pointer notation - rather than adding a integer offset `i` to the base address in `str`, we can copy the address in `str` into a pointer to a `char` and then *alter* that address to move from one element of the array to the next:

Version 3a: Pointer Arithmetic Notation

```
void string_to_upper(char* str)
{
    char* ptr;

    for (ptr = str; *ptr != '\0'; ptr++)
        *ptr = toupper(*ptr);
}
```

Okay, let's break down what's going on in this code:

- `str` is a pointer to a `char` (first character in the passed-in C string == first character in the `char` array)
- `ptr` is also a pointer to a `char`
- In the `for` loop, `ptr` is initialized to point to the same place `str` points to: copy the address stored in `str` into `ptr`.
- The loop continues as long as "the thing pointed to by `ptr`" is not the null character. That is, as long as we have not hit the end of the string
- Each time through the loop, "the `char` pointed to by `ptr`" is given to the `toupper()` function which returns the uppercase version of the `char`, and then that returned `char` is stored into "the `char` pointed to by `ptr`"
- Finally `ptr` is incremented so that it points to the next `char` in the array.

Of course, since `str` is itself a pointer, we can avoid using the extra variable `ptr` by just altering the address stored in `str`:

Version 3b: Pointer Arithmetic Notation

```
void string_to_upper(char* str)
{
    for (; *str != '\0'; str++)
        *str = toupper(*str);
}
```

If we recall that the null character has the ASCII value 0 and that an expression that resolves to 0 is false (while anything not 0 is true), we can shorten our code even further:

Version 4: The Final Version

```
void string_to_upper(char* str)
{
    for (; *str; str++)
        *str = toupper(*str);
}
```

As a C++ programmer, you can choose whichever representation suits you best. Most beginners prefer the subscript notation. However, understanding the pointer notation is important, if only for understanding code written by other programmers. For example, if you encounter a function written like this, you should be able to understand what it's doing::

```
size_t strlen(const char* str)
{
    const char* ptr;
    for (ptr = str; *ptr; ptr++)
        ;

    return ptr - str;
}
```

It's probably best not to mix notations unless you have a specific good reason to do so.