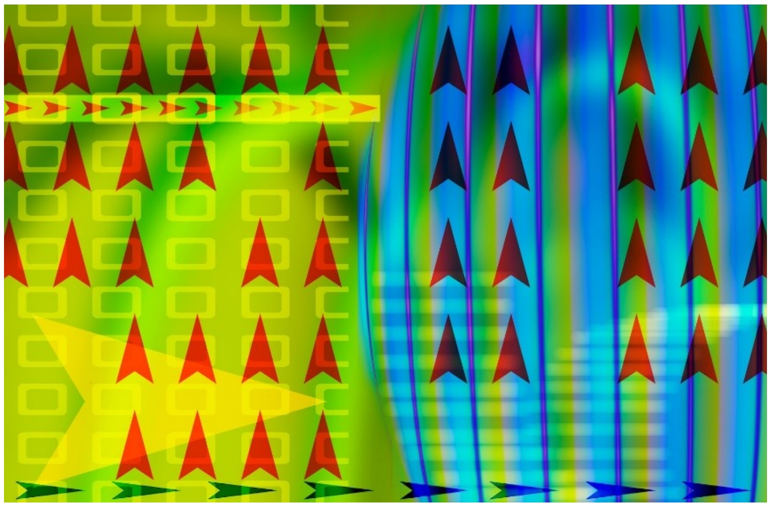
Pointers vs Array Names



In many ways, pointers are identical to array names. Both represent addresses of something in memory. However, there is a key difference between a pointer and an array name:

A pointer is a variable whose value (i.e., the address it points to) ***can*** *change, but an array name's value (i.e. the address it refers to)* ***cannot*** *change.*

Let's assume we have the following variable definitions:

```
int a[10] = { 11, 31, 51, 71, 91, 111, 131, 151, 171, 191 };
int b[] = { 63, 64, 65, 66, 67 };
int *p = a;      // both p and a now refer to the same place
int *p2 = b;     // both p2 and b now refer to the same place
```

Let's further assume that the array a is located in memory at address 0x323340 (where the 0x prefix indicates hexadecimal in C), the array b is located in memory at address 0x323368, the variable p is located in memory at address 0x320000, and the variable p2 is located in memory at address 0x320008. Also, assume that sizeof(int) is 4 bytes in our environment.

The table below shows a sampling of what is and isn't possible with pointers and array names. Notice that pointer arithmetic can be used with either pointers or array names.

Using a Pointer	Using an Array Name	Description	Expression Value	Note
`p`	`a`	address of 1st element of array	0x323340	base/starting address of the array
`&p[0]`	`&a[0]`	address of 1st element of array	0x323340	base/starting address of the array
`*p`	`*a`	contents of 1st element of array	11	1st element of the array
`p[0]`	`a[0]`	contents of 1st element of array	11	1st element of the array
`&p[1]`	`&a[1]`	address of 2nd element of array	0x323344	base address + sizeof(int) * 1
`p + 1`	`a + 1`	address of 2nd element of array	0x323344	base address + sizeof(int) * 1
`p[1]`	`a[1]`	contents of 2nd element of array	31	2nd element of the array
`&p[2]`	`&a[2]`	address of 3rd element of array	0x323348	base address + sizeof(int) * 2
`p + 2`	`a + 2`	address of 3rd element of array	0x323348	base address + sizeof(int) * 2
`p[2]`	`a[2]`	contents of 3rd element of array	51	3rd element of the array
`*(&p[2])`	`*(&a[2])`	contents of 3rd element of array	51	address of 3rd element of the array, dereferenced to get the data in that element
`*(p + 2)`	`*(a + 2)`	contents of 3rd element of array	51	address of 3rd element of the array, dereferenced to get the data in that element

`p = p + 1;`
`// or...`
`// p += 1;`
`// p++;`
`// ++p;`
		address of 2nd element of array	0x323344	Because of pointer arithmetic, incrementing by 1 adds 4 (the size of the data pointed to). We can't change address of an array.

`p = b;`		address of 1st element of b array	0x323368	We can set a pointer variable to point to the beginning of a completely different array. We can't change address of an array.
`p = &a[2];`		address of 3rd element of a array	0x323348	We can set a pointer variable to point to any element of an array. base address + sizeof(int) * 2. We can't change address of an array.
`p = &p2[3];`		address of 4th element of b array	0x323374	We can use array indexing with a pointer variable, as if it were an array name. base address + sizeof(int) * 3

`p = &p2[3];`
`p = &p[-1];`
		address of element just before the 4th element of b array (i.e., address of 3rd element of b array)	0x323370	In the 2nd assignment statement, p starts at the address of the 4th element of b. We then change p to point to the element just before that element (using the -1 index from the current position of p),

Using a Pointer	Using an Array Name	Description	Expression Value	Note
				which is the 3rd element of b. 1st assignment: base address + sizeof(int) * 3 2nd assignment: base address - sizeof(int) * 1
`&p`		address of pointer variable p	0x320000	The location in memory where the variable p is located. This has nothing to do with the address that happens to be stored inside the variable p (i.e., where p points).
`&p2`		address of pointer variable p2	0x320008	The location in memory where the variable p2 is located. This has nothing to do with the address that happens to be stored inside the variable p2 (i.e., where p2 points).

Note that a negative array index can be used with a pointer, as long as it doesn't end up accessing an element that is outside the bounds of the array (i.e. before the beginning of the array). If it does, undefined behavior (unpredictable behavior) will occur. The compiler can't help us if we index beyond either end of the array. It's up to our program logic to make sure we don't do that.