17.9 — Pointer arithmetic and subscripting

learncpp.com/cpp-tutorial/pointer-arithmetic-and-subscripting/

In lesson 16.1 -- Introduction to containers and arrays, we mentioned that arrays are stored sequentially in memory. In this lesson, we'll take a deeper look at how arrays indexing math works.

Although we won't use the indexing math in future lessons, the topics covered in this lesson will give you insight into how range-based for loops actually work, and will come in handy again later when we cover iterators.

What is pointer arithmetic?

Pointer arithmetic is a feature that allows us to apply certain integer arithmetic operators (addition, subtraction, increment, or decrement) to a pointer to produce a new memory address.

Given some pointer `ptr`, `ptr + 1` returns the address of the *next object* in memory (based on the type being pointed to). So if `ptr` is an `int*`, and an `int` is 4 bytes, `ptr + 1` will return the memory address that is 4 bytes after `ptr`, and `ptr + 2` will return the memory address that is 8 bytes after `ptr`.

```
#include <iostream>

int main()
{
    int x {};
    const int* ptr{ &x }; // assume 4 byte ints

    std::cout << ptr << ' ' << (ptr + 1) << ' ' << (ptr + 2) << '\n';

    return 0;
}
```

On the author's machine, this printed:

```
00AFFD80 00AFFD84 00AFFD88
```

Note that each memory address is 4 bytes greater than the previous.

Although less common, pointer arithmetic also works with subtraction. Given some pointer `ptr`, `ptr - 1` returns the address of the *previous object* in memory (based on the type being pointed to).

```
#include <iostream>

int main()
{
    int x {};
    const int* ptr{ &x }; // assume 4 byte ints

    std::cout << ptr << ' ' << (ptr - 1) << ' ' << (ptr - 2) << '\n';

    return 0;
}
```

On the author’s machine, this printed:

```
00AFFD80 00AFFD7C 00AFFD78
```

In this case, each memory address is 4 bytes less than the previous.

Key insight

Pointer arithmetic returns the address of the next/previous object (based on the type being pointed to), not the next/previous address.

Applying the increment (`++`) and decrement (`--`) operators to a pointer do the same thing as pointer addition and pointer subtraction respectively, but actually modify the address held by the pointer.

Given some int value `x`, `++x` is shorthand for `x = x + 1`. Similarly, given some pointer `ptr`, `++ptr` is shorthand for `ptr = ptr + 1`, which does pointer arithmetic and assigns the result back to `ptr`.

```
#include <iostream>

int main()
{
    int x {};
    const int* ptr{ &x }; // assume 4 byte ints

    std::cout << ptr << '\n';

    ++ptr; // ptr = ptr + 1
    std::cout << ptr << '\n';

    --ptr; // ptr = ptr - 1
    std::cout << ptr << '\n';

    return 0;
}
```

On the author’s machine, this printed:

```
00AFFD80 00AFFD84 00AFFD80
```

Subscripting is implemented via pointer arithmetic

In the prior lesson (17.8 -- C-style array decay), we noted that `operator[]` can be applied to a pointer:

```
#include <iostream>

int main()
{
    const int arr[] { 9, 7, 5, 3, 1 };

    const int* ptr{ arr }; // a normal pointer holding the address of element 0
    std::cout << ptr[2];   // subscript ptr to get element 2, prints 5

    return 0;
}
```

Let's take a deeper look at what's happening here.

It turns out that subscript operation `ptr[n]` is a concise syntax equivalent to the more verbose expression `*((ptr) + (n))`. You'll note that this is just pointer arithmetic, with some additional parenthesis to ensure things evaluate in the correct order, and an implicit dereference to get the object at that address.

First, we initialize `ptr` with `arr`. When `arr` is used as an initializer, it decays into a pointer holding the address of the element with index 0. So `ptr` now holds the address of element 0.

Next, we print `ptr[2]`. `ptr[2]` is equivalent to `*((ptr) + (2))`, which is equivalent to `*(ptr + 2)`. `ptr + 2` returns the address of the object that is two objects past `ptr`, which is the element with index 2. The object at that address is then returned to the caller.

Let's take a look at another example:

```
#include <iostream>

int main()
{
    const int arr[] { 3, 2, 1 };

    // First, let's use subscripting to get the address and values of our array
elements
    std::cout << &arr[0] << ' ' << &arr[1] << ' ' << &arr[2] << '\n';
    std::cout << arr[0] << ' ' << arr[1] << ' ' << arr[2] << '\n';

    // Now let's do the equivalent using pointer arithmetic
    std::cout << arr<< ' ' << (arr+ 1) << ' ' << (arr+ 2) << '\n';
    std::cout << *arr<< ' ' << *(arr+ 1) << ' ' << *(arr+ 2) << '\n';

    return 0;
}
```

On the author’s machine, this printed:

```
00AFFD80 00AFFD84 00AFFD88
3 2 1
00AFFD80 00AFFD84 00AFFD88
3 2 1
```

You’ll note that `ptr` is holding address `00AFFD80`, `(ptr + 1)` returns an address 4 bytes later, and `(ptr + 2)` returns an address 8 bytes later. We can dereference these addresses to get the elements at those addresses.

Because array elements are always sequential in memory, if `ptr` is a pointer to element 0 of an array, `*(ptr + n)` will return the n-th element in the array.

This is the primary reason arrays are 0-based rather than 1-based. It makes the math more efficient (because the compiler doesn’t have to subtract 1 whenever subscripting)!

As an aside…

As a neat bit of trivia, because the compiler converts `ptr[n]` into `*((ptr) + (n))` when subscripting a pointer, this means we can also subscript a pointer as `n[ptr]`! The compiler converts this into `*((n) + (ptr))`, which is behaviorally identical to `*((ptr) + (n))`. Don’t actually do this though, as it’s confusing.

Pointer arithmetic and subscripting are relative addresses

When first learning about array subscripting, it’s natural to assume that the index represents a fixed element within the array: Index 0 is always the first element, index 1 is always the second element, etc…

This is a illusion. Array indices are actually relative positions. The indices just appear fixed because we almost always index from the start (element 0) of the array!

Remember, given some pointer `ptr`, both `*(ptr + 1)` and `ptr[1]` return the *next object* in memory (based on the type being pointed to). Next is a relative term, not an absolute one. Thus, if `ptr` is pointing to element 0, then both `*(ptr + 1)` and `ptr[1]` will return element 1. But if `ptr` is pointing to element 3 instead, then both `*(ptr + 1)` and `ptr[1]` will return element 4!

The following example demonstrates this:

```
#include <array>
#include <iostream>

int main()
{
    const int arr[] { 9, 8, 7, 6, 5 };
    const int *ptr { arr }; // arr decays into a pointer to element 0

    // Prove that we're pointing at element 0
    std::cout << *ptr << ptr[0] << '\n'; // prints 99
    // Prove that ptr[1] is element 1
    std::cout << *(ptr+1) << ptr[1] << '\n'; // prints 88

    // Now set ptr to point at element 3
    ptr = &arr[3];

    // Prove that we're pointing at element 3
    std::cout << *ptr << ptr[0] << '\n'; // prints 66
    // Prove that ptr[1] is element 4!
    std::cout << *(ptr+1) << ptr[1] << '\n'; // prints 55

    return 0;
}
```

However, you'll also note that our program is a lot more confusing if we can't assume that `ptr[1]` is always the element with index 1. For this reason, we recommend using subscripting only when indexing from the start of the array (element 0). Use pointer arithmetic only when doing relative positioning.

Best practice

Favor subscripting when indexing from the start of the array (element 0), so the array indices line up with the element.

Favor pointer arithmetic when doing relative positioning from a given element.

Negative indices

In the last lesson, we mentioned that (unlike the standard library container classes) the index of a C-style array can be either an unsigned integer or a signed integer. This wasn't done just for convenience -- it's actually possible to index a C-style array with a negative subscript. It sounds funny, but it makes sense.

We just covered that `*(ptr+1)` returns the *next object* in memory. And `ptr[1]` is just a convenient syntax to do the same.

At the top of this lesson, we noted that `*(ptr-1)` returns the *previous object* in memory. Want to guess what the subscript equivalent is? Yup, `ptr[-1]`.

```
#include <array>
#include <iostream>

int main()
{
    const int arr[] { 9, 8, 7, 6, 5 };

    // Set ptr to point at element 3
    const int* ptr { &arr[3] };

    // Prove that we're pointing at element 3
    std::cout << *ptr << ptr[0] << '\n'; // prints 66
    // Prove that ptr[-1] is element 2!
    std::cout << *(ptr-1) << ptr[-1] << '\n'; // prints 77

    return 0;
}
```

Pointer arithmetic can be used to traverse an array

One of the most common uses of pointer arithmetic is to iterate through a C-style array without explicit indexing. The following example illustrates how this is done:

```
#include <iostream>

int main()
{
        constexpr int arr[]{ 9, 7, 5, 3, 1 };

        const int* begin{ arr };                // begin points to start element
        const int* end{ arr + std::size(arr) }; // end points to one-past-the-end
element

        for (; begin != end; ++begin)           // iterate from begin up to (but
excluding) end
        {
                std::cout << *begin << ' ';     // dereference our loop variable to
get the current element
        }

        return 0;
}
```

In the above example, we start our traversal at the element pointed to by `begin` (which in this case is element 0 of the array). Since `begin != end` yet, the loop body executes. Inside the loop, we access the current element via `*begin`, which is just a pointer dereference. After the loop body, we do `++begin`, which uses pointer arithmetic to increment `begin` to point at the next element. Since `begin != end`, the loop body executes again. This continues until `begin != end` is `false`, which happens when `begin == end`.

Thus, the above prints:

```
9 7 5 3 1
```

Note that `end` is set to one-past-the-end of the array. Having `end` hold this address is fine (so long as we don't dereference `end`, as there isn't a valid element at that address). We do this because it makes our math and comparisons as simple as possible (no need to add or subtract 1 anywhere).

Tip

For a pointer that is pointing to a C-style array element, pointer arithmetic is valid so long as the resulting address is the address of a valid array element, or one-past the last element. If pointer arithmetic results in an address beyond these bounds, it is undefined behavior (even if the result is not dereferenced).

In the prior lesson 17.8 -- C-style array decay, we mentioned that array decay makes refactoring functions difficult because certain things work with non-decayed arrays but not with decayed arrays (like `std::size`). One neat thing about this traversing an array this way is that we can refactor the loop part of the above example into a separate function exactly as written, and it will still work:

```
#include <iostream>

void printArray(const int* begin, const int* end)
{
	for (; begin != end; ++begin)   // iterate from begin up to (but excluding) end
	{
		std::cout << *begin << ' '; // dereference our loop variable to get the current element
	}

	std::cout << '\n';
}

int main()
{
	constexpr int arr[]{ 9, 7, 5, 3, 1 };

	const int* begin{ arr };                // begin points to start element
	const int* end{ arr + std::size(arr) }; // end points to one-past-the-end element

	printArray(begin, end);

	return 0;
}
```

Note that this program compiles and produces the correct result even though we never explicitly pass the array to the function! And because we're not passing `arr`, we don't have to deal with a decayed `arr` in `printArray()`. Instead, `begin` and `end` contain all the information we need to traverse the array.

In future lessons (when we cover iterators and algorithms), we'll see that the standard library is full of functions that use a `begin` and `end` pair to define what elements of a container the function should operate on.

Range-based for loops over C-style arrays are implemented using pointer arithmetic

Consider the following range-based for loop:

```
#include <iostream>

int main()
{
        constexpr int arr[]{ 9, 7, 5, 3, 1 };

        for (auto e : arr)         // iterate from `begin` up to (but excluding)
`end`
        {
                std::cout << e << ' '; // dereference our loop variable to get the
current element
        }

        return 0;
}
```

If you look at the documentation for range-based for loops, you’ll see that they are typically implemented something like this:

```
{
    auto __begin = begin-expr;
    auto __end = end-expr;

    for ( ; __begin != __end; ++__begin)
    {
        range-declaration = *__begin;
        loop-statement;
    }
}
```

Let’s replace the range-based for loop in the prior example with this implementation:

```
#include <iostream>

int main()
{
        constexpr int arr[]{ 9, 7, 5, 3, 1 };

        auto __begin = arr;                // arr is our begin-expr
        auto __end = arr + std::size(arr); // arr + std::size(arr) is our end-expr

        for ( ; __begin != __end; ++__begin)
        {
                auto e = *__begin;         // e is our range-declaration
                std::cout << e << ' ';     // here is our loop-statement
        }

        return 0;
}
```

Note how similar this is to the example we wrote in the prior section! The only difference is that we're assigning `*__begin` to `e` and using `e` rather than just using `*__begin` directly!

Quiz time

Question #1

a) Why is `arr[0]` the same as `*arr`?

Show Solution

Related content

We have more quiz questions on pointer arithmetic in the next lesson (17.10 -- C-style strings).