10.11 — Pointer arithmetic and array indexing

ALEX JULY 20, 2021

Pointer arithmetic

The C++ language allows you to perform integer addition or subtraction operations on pointers. If `ptr` points to an integer, `ptr + 1` is the address of the next integer in memory after ptr. `ptr - 1` is the address of the previous integer before `ptr`.

Note that `ptr + 1` does not return the *memory address* after `ptr`, but the memory address of the *next object of the type* that `ptr` points to. If `ptr` points to an integer (assuming 4 bytes), `ptr + 3` means 3 integers (12 bytes) after `ptr`. If `ptr` points to a `char`, which is always 1 byte, `ptr + 3` means 3 chars (3 bytes) after ptr.

When calculating the result of a pointer arithmetic expression, the compiler always multiplies the integer operand by the size of the object being pointed to. This is called scaling.

Consider the following program:

```
#include <iostream>

int main()
{
    int value{ 7 };
    int* ptr{ &value };

    std::cout << ptr << '\n';
    std::cout << ptr+1 << '\n';
    std::cout << ptr+2 << '\n';
    std::cout << ptr+3 << '\n';

    return 0;
}
```

On the author's machine, this output:

```
0012FF7C
0012FF80
0012FF84
0012FF88
```

As you can see, each of these addresses differs by 4 (7C + 4 = 80 in hexadecimal). This is because an integer is 4 bytes on the author's machine.

The same program using `short` instead of `int`:

```
#include <iostream>

int main()
{
    short value{ 7 };
    short* ptr{ &value };

    std::cout << ptr << '\n';
    std::cout << ptr+1 << '\n';
    std::cout << ptr+2 << '\n';
    std::cout << ptr+3 << '\n';

    return 0;
}
```

On the author's machine, this output:

```
0012FF7C
0012FF7E
0012FF80
0012FF82
```

Because a short is 2 bytes, each address differs by 2.

Arrays are laid out sequentially in memory

By using the address-of operator (&), we can determine that arrays are laid out sequentially in memory. That is, elements 0, 1, 2, ... are all adjacent to each other, in order.

```
#include <iostream>

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

    std::cout << "Element 0 is at address: " << &array[0] << '\n';
    std::cout << "Element 1 is at address: " << &array[1] << '\n';
    std::cout << "Element 2 is at address: " << &array[2] << '\n';
    std::cout << "Element 3 is at address: " << &array[3] << '\n';

    return 0;
}
```

On the author's machine, this printed:

```
Element 0 is at address: 0041FE9C
Element 1 is at address: 0041FEA0
Element 2 is at address: 0041FEA4
Element 3 is at address: 0041FEA8
```

Note that each of these memory addresses is 4 bytes apart, which is the size of an integer on the author's machine.

Pointer arithmetic, arrays, and the magic behind indexing

In the section above, you learned that arrays are laid out in memory sequentially.

In the previous lesson, you learned that a fixed array can decay into a pointer that points to the first element (element 0) of the array.

Also in a section above, you learned that adding 1 to a pointer returns the memory address of the next object of that type in memory.

Therefore, we might conclude that adding 1 to an array should point to the second element (element 1) of the array. We can verify experimentally that this is true:

```
#include <iostream>

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

     std::cout << &array[1] << '\n'; // print memory address of array element 1
     std::cout << array+1 << '\n'; // print memory address of array pointer + 1

     std::cout << array[1] << '\n'; // prints 7
     std::cout << *(array+1) << '\n'; // prints 7 (note the parenthesis required here)

    return 0;
}
```

Note that when performing indirection through the result of pointer arithmetic, parenthesis are necessary to ensure the operator precedence is correct, since operator * has higher precedence than operator +.

On the author's machine, this printed:

```
0017FB80
0017FB80
7
7
```

It turns out that when the compiler sees the subscript operator ([]), it actually translates that into a pointer addition and indirection! Generalizing, `array[n]` is the same as `*(array + n)`, where n is an integer. The subscript operator [] is there both to look nice and for ease of use (so you don't have to remember the parenthesis).

Using a pointer to iterate through an array

We can use a pointer and pointer arithmetic to loop through an array. Although not commonly done this way (using subscripts is generally easier to read and less error prone), the following example goes to show it is possible:

```cpp
#include <iostream>
#include <iterator> // for std::size

bool isVowel(char ch)
{
    switch (ch)
    {
    case 'A':
    case 'a':
    case 'E':
    case 'e':
    case 'I':
    case 'i':
    case 'O':
    case 'o':
    case 'U':
    case 'u':
        return true;
    default:
        return false;
    }
}

int main()
{
    char name[]{ "Mollie" };
    int arrayLength{ static_cast<int>(std::size(name)) };
    int numVowels{ 0 };

    for (char* ptr{ name }; ptr != (name + arrayLength);
++ptr)
    {
        if (isVowel(*ptr))
        {
            ++numVowels;
        }
    }

    std::cout << name << " has " << numVowels << "
vowels.\n";

    return 0;
}
```

How does it work? This program uses a pointer to step through each of the elements in an array. Remember that arrays decay to pointers to the first element of the array. So by assigning `ptr` to name, `ptr` will also point to the first element of the array. Indirection through `ptr` is performed for each element when we call `isVowel(*ptr)`, and if the element is a vowel, `numVowels` is incremented. Then the for loop uses the ++ operator to advance the pointer to the next character in the array. The for loop terminates when all characters have been examined.

The above program produces the result:

```
Mollie has 3 vowels
```

Because counting elements is common, the algorithms library offers `std::count_if`, which counts elements that fulfill a condition. We can replace the `for`-loop with a call to `std::count_if`.

```cpp
#include <algorithm>
#include <iostream>
#include <iterator> // for std::begin and std::end

bool isVowel(char ch)
{
    switch (ch)
    {
    case 'A':
    case 'a':
    case 'E':
    case 'e':
    case 'I':
    case 'i':
    case 'O':
    case 'o':
    case 'U':
    case 'u':
        return true;
    default:
        return false;
    }
}

int main()
{
    char name[]{ "Mollie" };

    // walk through all the elements of name and count how many calls to isVowel return true
    auto numVowels{ std::count_if(std::begin(name), std::end(name), isVowel) };

    std::cout << name << " has " << numVowels << " vowels.\n";

    return 0;
}
```

`std::begin` returns an iterator (pointer) to the first element, while `std::end` returns an iterator to the element that would be one after the last. The iterator returned by `std::end` is only used as a marker, accessing it causes undefined behavior, because it doesn't point to a real element.

`std::begin` and `std::end` only work on arrays with a known size. If the array decayed to a pointer, we can calculate begin and end manually.

```cpp
// nameLength is the number of elements in the array.
std::count_if(name, name + nameLength, isVowel)

// Don't do this. Accessing invalid indexes causes undefined behavior.
// std::count_if(name, &name[nameLength], isVowel)
```

Note that we're calculating `name + nameLength`, not `name + nameLength - 1`, because we don't want the last element, but the pseudo-element one past the last.

Calculating begin and end of an array like this works for all algorithms that need a begin and end argument.

Quiz time

Question #1

Why does the following code work?

```cpp
#include <iostream>

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

 std::cout << 2[arr] << '\n';

 return 0;
}
```

Show Solution

Question #2

Write a function named `find` that takes a pointer to the beginning and a pointer to the end (1 element past the last) of an array, as well as a value. The function should search for the given value and return a pointer to the first element with that value, or the end pointer if no element was found. The following program should run:

```
#include <iostream>
#include <iterator>

// ...

int main()
{
    int arr[]{ 2, 5, 4, 10, 8, 20, 16, 40 };

    // Search for the first element with value 20.
    int* found{ find(std::begin(arr), std::end(arr), 20)
};

    // If an element with value 20 was found, print it.
    if (found != std::end(arr))
    {
        std::cout << *found << '\n';
    }

    return 0;
}
```

Tip

`std::begin` and `std::end` return an `int*`. The call to `find` is equivalent to

```
int* found{ find(arr, arr + std::size(arr), 20)
};
```

Show Solution

Next lesson

10.12 C-style string symbolic constants

Back to table of contents

Previous lesson

10.10 Pointers and arrays

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest