6.8a — Pointer arithmetic and array indexing

BY ALEX ON AUGUST 15TH, 2015 | LAST MODIFIED BY ALEX ON JULY 24TH, 2017

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 lesson **6.8 -- Pointers and arrays**, 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 [5] = { 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 dereferencing 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 dereference! 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:

```
const int arrayLength = 7;
char name[arrayLength] = "Mollie";
int numVowels(0);
for (char *ptr = name; ptr < name + arrayLength; ++ptr)
{
    switch (*ptr)
    {
        case 'A':
        case 'a':
        case 'E':
        case 'e':
        case 'I':
        case 'i':
        case 'O':
        case 'o':
        case 'U':
        case 'u':
            ++numVowels;
    }
}

cout << name << " has " << numVowels << " vowels.\n";
```

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. Each element is dereferenced by the switch expression, 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
```



6.8b -- C-style string symbolic constants


Index


6.8 -- Pointers and arrays