6.8a — Pointer arithmetic and array indexing

BY ALEX ON AUGUST 15TH, 2015 | LAST MODIFIED BY NASCARDRIVER ON JANUARY 17TH, 2020

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 []{ 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:

```
#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. 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
```

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`.

```
#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" };
  auto numVowels{ std::count_if(std::begin(name), std::end(name), isVowel) };

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

```

```
32    return 0;
33  }
```

`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.

```
1  // nameLength is the number of elements in the array.
2  std::count_if(name, name + nameLength, isVowel)
3
4  // Don't do this. Accessing invalid indexes causes undefined behavior.
5  // std::count_if(name, &name[nameLength], isVowel)
```

Quiz time

Question #1

Why does the following code work?

```
1  #include <iostream>
2
3  int main()
4  {
5    int arr[]{ 1, 2, 3 };
6
7    std::cout << 2[arr] << '\n';
8
9    return 0;
10 }
```

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:

```
1  #include <iostream>
2  #include <iterator>
3
4  // ...
5
6  int main()
7  {
8      int arr[]{ 2, 5, 4, 10, 8, 20, 16, 40 };
9
10     // Search for the first element with value 20.
11     int* found{ find(std::begin(arr), std::end(arr), 20) };
12
13     // If an element with value 20 was found, print it.
14     if (found != std::end(arr))
15     {
16         std::cout << *found << '\n';
17     }
18
19     return 0;
20 }
```

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


6.8b -- C-style string symbolic constants


Index


6.8 -- Pointers and arrays

C++ TUTORIAL | PRINT THIS POST

189 comments to 6.8a — Pointer arithmetic and array indexing

« Older Comments 1 2 3

kavin
January 23, 2020 at 8:36 am · Reply

For quiz 2 i did this.

```
int* find(int* begin, int* end, int value)
{
    for (begin; begin < end; ++begin)
    {
        if (*begin == value)
```

```
        {
            return begin;
        }
    }
    return end;
}
```

I saw your solution. Why are you initializing begin to pointer p? we already know begin is a pointer right?can't we use begin as a address directly? and can i use begin < end , since we know end is " arr + std::size(arr) " and compare 2 addresses to loop through them?

nascardriver
January 23, 2020 at 8:52 am · Reply

> Why are you initializing begin to pointer p?
Modifying arguments makes code harder to understand. Once you modify `begin`, it's name is no longer correct, it's not pointing to the beginning of the data.

> can i use begin < end Yes, there's nothing wrong with it. Using != is more portable, you'll learn more about it in the lesson about iterators.

kavin
January 23, 2020 at 9:12 am · Reply

Oh ok, thank you. Now i get it. Since we were using variable values in the called function in previous chapters without initializing them to other variable , i thought i could apply it here. So for pointers the best practice is not modify the parameter and initialize them to another pointer and modify that ?

And i have another common doubt for quite sometime.I forgot to ask you in previous chapters. Why do u use "return end; " outside of loop? Can't you put an else and use like this,

```
int* find(int* begin, int* end, int value)
{
    for (int* p{begin}; p != end; ++p)
    {
        if (*p == value)
            return p;
        else
            return end;
    }
}
```

nascardriver
January 23, 2020 at 9:15 am · Reply

Modifying parameters is never good if it changes the meaning of the variable. Your suggested update doesn't work. The function always returns in the first iteration.

kavin
January 23, 2020 at 9:54 am · Reply

>The function always returns in the first iteration<

Oh yes ! I couldn't figure out this issue small issue till now :(Thank you @nascardriver.

Suyash
January 20, 2020 at 5:23 am · Reply

Here's the code to my solution...

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

int* find(int *begin, int *end, int value)
{
    for (int* ptr{ begin }; ptr != end; ++ptr)
        if (*ptr == value)
            return ptr;

    return end;
}
```

nascardriver
January 20, 2020 at 5:30 am · Reply

Pretty much exactly the same as the solution, good job!

chai
December 31, 2019 at 6:52 am · Reply

[code]
std::count_if(name, name + nameLength, isVowel)
[code]
What type of functions are compatible with std::count_if? Something similar to bool isVowel(char), I am guessing that it has to accept a type to be counted and returns a bool and count_if counts the bool returned? .It is also odd and first encounter of a function being passed as argument without ().

nascardriver
December 31, 2019 at 7:01 am · Reply

The function has to return a `bool` and take an argument of the array element type. We have an array of `char`s, so the function has to accept a char. It could also accept types that can be created from a char, eg. an `int`.
If we had an array of `std::string`s, the the function would have to accept a `std::string`.

Algorithms will get more prominent in the tutorials. There will be a short introduction about them once it's clear in which lesson they're first used. It's not too difficult to understand how to use algorithms without knowing how they work, so they're adding them to the tutorials before we explain how they work.

chai
December 30, 2019 at 1:26 pm · Reply

It would be great to have an exercise here.

nascardriver
January 17, 2020 at 6:59 am · Reply

Thank your for your feedback!

I added two questions to the lesson to help future readers understanding pointer arithmetic.

elvis
December 3, 2019 at 2:24 pm · Reply

in the example where it sorts through mollie to find how many vowels there are. the for loop has this condition in it ptr < (name + arrayLength) how does it add name and int? I tried printing "name + arrayLength" and i get random garbage. does name and array length convert to pointers when they are being compared to ptr?

nascardriver
December 4, 2019 at 4:11 am · Reply

Hi Elvis!

Your question is the main topic of this lesson, I suggest you re-read it.
`name + arrayLength` returns a pointer that points `arrayLength` bytes after `name`.

Elvis
December 13, 2019 at 8:50 am · Reply

Had a bit of a brain fart.
Thank you nascar driver!

Ged
November 20, 2019 at 11:36 am · Reply

Code is missing the <algorithm> library. And why are we using std::size_t, because when I try to run it I get an error and if I change it to an int, it works. Only if I use the static_cast<std::size_t>( ) it works, but why do the extra work? As I understand size_t is an unsigned int which you told to avoid if you can.

```
#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;
  }
}
```

```
23
24  int main()
25  {
26    char name[]{ "Mollie" };
27    std::size_t numVowels{ std::count_if(std::begin(name), std::end(name), isVowel) };
28
29    std::cout << name << " has " << numVowels << " vowels.\n";
30
31    return 0;
32  }
```

nascardriver
November 21, 2019 at 2:43 am · Reply

> Code is missing the library
Added.

> why are we using std::size_t
I thought `std::count_if` returned an `std::size_t`. It returns a `std::ptrdiff_t` in this case. Code updated to use `auto`.

> As I understand size_t is an unsigned int which you told to avoid if you can.
It's an unsigned integer, but not necessarily an unsigned int. If you don't modify an unsigned integer and don't use it for arithmetic, there's nothing that can go wrong. If `std::count_if` returned a `std::size_t`, we would've needed an extra cast when all we want to do is print the result.

Thanks!

Ged
November 20, 2019 at 10:51 am · Reply

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).

Isn't [] used to save a changed value which *() can't do?

```
1  array[n] = 7; this will work
2  *(array + n) = 7; this shouldn't work?
```

nascardriver
November 21, 2019 at 2:45 am · Reply

Indirection (*ptr) returns a reference (covered later). References can be used to modify the value.

Jack Overby
October 28, 2019 at 9:48 am · Reply

I'm trying to loop through and perform a regex check on each character, rather than switch-10 different cases:

```
1  for (char *ptr = name; ptr < name + chlen; ++ptr)
2      {
3          std::cout << *ptr << "\n";
4          if (std::regex_match(*ptr, std::regex("[aeiouAEIOU]")))
```

```
        {
            ++numVowels;
        }
    }
```

However, I keep getting the following compiler error message:

no instance of overloaded function "std::regex_match" matches the argument list

Any suggestions?

nascardriver
October 29, 2019 at 2:21 am · Reply

Regex shouldn't be used for this. It has a huge overhead compared to manually solving the task.
You'll get the best results from using a `switch`. I understand you don't want to do that.
You can use an `std::set` and `std::count_if` instead.

```
#include <algorithm>
#include <iostream>
#include <set>
#include <string>

int main(void)
{
  std::string strTest{ "Hello AJack" };
  std::set setVowels{ 'a', 'e', 'i', 'o', 'u',
                      'A', 'E', 'I', 'O', 'U' };

  // You could add AEIOU to @strVowels automatically.
  // std::transform(setVowels.begin(), setVowels.end(),
  //                std::inserter(setVowels, setVowels.end()), [](char ch) {
  //                  return std::toupper(ch);
  //                });

  // The [](){} is an anonymous function (Lambda).
  auto sVowels{ std::count_if(strTest.begin(), strTest.end(), [&](char ch) {
    return (setVowels.count(ch) > 0); // setVowels.contains(ch) since C++20
  }) };

  std::cout << sVowels << " vowels\n";

  return 0;
}
```

If you really wanted to use regex, which you shouldn't, you could use an `std::sregex_iterator` and `std::distance`. `std::sregex_iterator` iterates over all found matches.

```
#include <iostream>
#include <iterator>
#include <regex>
#include <string>

int main(void)
{
  std::string strTest{ "Hello Jack" };
  std::regex rxVowels{ "[aeiou]", std::regex::icase };

  auto itBegin{
    std::sregex_iterator{ strTest.begin(), strTest.end(), rxVowels }
  };
```

```
14
15      std::cout << std::distance(itBegin, std::sregex_iterator{}) << " vowels\n";
16
17      return 0;
18  }
```

alfonso
September 26, 2019 at 1:28 am · Reply

Here the char array name does not decay or std::cout treats it in a special way.

```
1  #include <iostream>
2
3  int main () {
4      char name [] {"Alberta"};
5
6      std::cout << name << '\n';
7      std::cout << name + 1 << '\n';
8      std::cout << name + 2 << '\n';
9      std::cout << name + 3 << '\n';
10
11     return 0;
12 }
```

And maybe for the same reason, the following code gives me strange results:

```
1  #include <iostream>
2
3  int main () {
4      char ch {'a'};
5      char *chptr {&ch};
6
7      std::cout << chptr << '\n';
8      std::cout << chptr + 1 << '\n';
9      std::cout << chptr + 2 << '\n';
10     std::cout << chptr + 3 << '\n';
11
12     return 0;
13 }
```

nascardriver
September 26, 2019 at 4:05 am · Reply

std::cout treats it in a special way

« Older Comments 1 2 3