Pointer Arithmetic

Does anyone have any good articles or explanations (blogs, examples) for pointer arithmetic? Figure the audience is a bunch of Java programmers learning C and C++.

c pointers pointer-arithmetic

edited Feb 17 '09 at 20:26
starblue
37k 9 62 114

asked Dec 27 '08 at 7:10
leora
15.2k 224 615 1093

1 c-faq.com section 4 – Keith Thompson Oct 13 '15 at 20:29

7 Answers

First, the binky video may help. It's a nice video about pointers. For arithmetic, here is an example:

```
int * pa = NULL;
int * pb = NULL;
pa += 1; // pa++. behind the scenes, add sizeof(int) bytes
assert((pa - pb) == 1);

print_out(pa); // possibly outputs 0x4
print_out(pb); // possibly outputs 0x0 (if NULL is actually bit-wise 0x0)
```

(Note that incrementing a pointer that contains a null pointer value strictly is undefined behavior. We used NULL because we were only interested in the value of the pointer. Normally, only use increment/decrement when pointing to elements of an array).

The following shows two important concepts

- addition/subtraction of a integer to a pointer means move the pointer forward / backward by N elements. So if an int is 4 bytes big, pa could contain 0x4 on our platform after having incremented by 1.
- subtraction of a pointer by another pointer means getting their distance, measured by elements. So subtracting pb from pa will yield 1, since they have one element distance.

On a practical example. Suppose you write a function and people provide you with an start and end pointer (very common thing in C++):

```
void mutate_them(int *begin, int *end) {
    // get the amount of elements
    ptrdiff_t n = end - begin;
    // allocate space for n elements to do something...
    // then iterate. increment begin until it hits end
    while(begin != end) {
        // do something
        begin++;
    }
}
```

`ptrdiff_t` is what is the type of (end - begin). It may be a synonym for "int" for some compiler, but may be another type for another one. One cannot know, so one chooses the generic typedef `ptrdiff_t`.

edited Dec 27 '08 at 10:18

answered Dec 27 '08 at 7:21
Johannes Schaub - litb
329k 71 648 1017

It should be noted that <stddef.h> or <cstddef> defines ptrdiff_t (per the standard). It is not a special type, I would say, just a name (typedef) for the type the compiler spits out. – strager Dec 27 '08 at 7:46

yeah indeed, must be some signed integer type. didn't want to go too much into details. – Johannes Schaub - litb Dec 27 '08 at 7:50

7 Hate to point it out, but your first example is undefined behavior. ;) You're not allowed to increment a null pointer. :) – jalf Dec 27 '08 at 9:52

jalf, thanks for pointing that out. i added a note about it :) well now my saying "if NULL is actually bitwise 0" is broken, since 0 is a zero integer. but i think u know what i mean hehe – Johannes Schaub - litb Dec 27 '08 at 10:41

Here is where I learned pointers: http://www.cplusplus.com/doc/tutorial/pointers.html

Once you understand pointers, pointer arithmetic is easy. The only difference between it and regular arithmetic is that the number you are adding to the pointer will be multiplied by the size of the type that the pointer is pointing to. For example, if you have a pointer to an `int` and an `int`'s size is 4 bytes, `(pointer_to_int + 4)` will evaluate to a memory address 16 bytes (4 ints) ahead.

So when you write

```
(a_pointer + a_number)
```

in pointer arithmetic, what's really happening is

```
(a_pointer + (a_number * sizeof(*a_pointer)))
```

in regular arithmetic.

answered Dec 27 '08 at 7:31
Jeremy Ruten
88.7k 25 131 171

1 Very concise and well put. – Anthony Giorgio Dec 29 '08 at 21:27

But a Pointer address can't be added yo another one. Why – Martin Erhardt Dec 16 '12 at 22:49

Sorry to be the 13th upvote! But this answer is great, explained well. – SSH This Feb 19 '13 at 23:00

The article linked to is great, makes sooo much more sense now! – CWitty Jan 8 '15 at 4:27

@MartinErhardt, you can like this p = (T*)(((long)p) + ((long)p2)) – Great.And.Powerful.Oz Oct 23 '15 at 16:41

---

applying NLP, call it address arithmetic. 'pointers' are feared and misunderstood mostly because they are taught by the wrong people and/or at the wrong stage with wrong examples in the wrong way. It is no wonder that nobody 'gets' it.

when teaching pointers, the faculty goes on about "p is a pointer to a, the value of p is the address of a" and so on. it just wont work. here is the raw material for you to build with. practice with it and your students will get it.

'int a', a is an integer, it stores integer type values. 'int* p', p is an 'int star', it stores 'int star' type values.

'a' is how you get the 'what' integer stored in a (try not to use 'value of a') '&a' is how you get the 'where' a itself is stored (try to say 'address') 'b = a' for this to work, both sides must be of the same type. if a is int, b must be capable of storing an int. (so __ b, the blank is filled with 'int')

'p = &a' for this to work, both sides must be of the same type. if a is an integer, &a is an address, p must be capable of storing addresses of integers. (so __ p, the blank is filled with 'int *')

now write int *p differently to bring out the type information:

int* | p

what is 'p'? ans: it is 'int *'. so 'p' is an address of an integer.

int | *p

what is '*p'? ans: it is an 'int'. so '*p' is an integer.

now on to the address arithmetic:

int a; a=1; a=a+1;

what are we doing in 'a=a+1'? think of it as 'next'. Because a is a number, this is like saying 'next number'. Since a holds 1, saying 'next' will make it 2.

// fallacious example. you have been warned!!! int *p int a; p = &a; p=p+1;

what are we doing in 'p=p+1'? it is still saying 'next'. This time, p is not a number but an address. So what we are saying is 'next address'. Next address depends on the data type, more specifically on the size of the data type.

printf("%d %d %d", sizeof(char), sizeof(int), sizeof(float));

so 'next' for an address will move forward sizeof(data type).

this has worked for me and all of the people I used to teach.

answered Dec 27 '08 at 8:08

Kinjal Dixit

I consider a good example of pointer arithmetic the following string length function:

```
int length(char *s)
{
   char *str = s;
   while(*str++);
   return str - s;
}
```

answered Dec 27 '08 at 7:44

arul
11.8k 1 40 73

So, the key thing to remember is that a pointer is just a word-sized variable that's typed for dereferencing. That means that whether it's a void *, int *, long long **, it's still just a word sized variable. The difference between these types is what the compiler considers the dereferenced type. Just to clarify, word sized means width of a virtual address. If you don't know what this means, just remember on a 64-bit machine, pointers are 8 bytes, and on a 32-bit machine, pointers are 4 bytes. The concept of an address is SUPER important in understanding pointers. An address is a number capable of uniquely identifying a certain location in memory. Everything in memory has an address. For our purposes, we can say that every variable has an address. This isn't necessarily always true, but the compiler lets us assume this. The address itself is byte granular, meaning 0x0000000 specifies the beginning of memory, and 0x00000001 is one byte into memory. This means that by adding one to a pointer, we're moving one byte forward into memory. Now, lets take arrays. If you create an array of type quux that's 32 elements big, it will span from the beginning of it's allocation, to the beginning of it's allocation plus 32*sizeof(quux), since each cell of the array is sizeof(quux) big. So, really when we specify an element of an array with array[n], that's just syntactic sugar (shorthand) for *(array+sizeof(quux)*n). Pointer arithmetic is really just changing the address that you're referring to, which is why we can implement strlen with

```
while(*n++ != '\0'){
  len++;
}
```

since we're just scanning along, byte by byte until we hit a zero. Hope that helps!

answered Dec 27 '08 at 8:11

gaze

There are several ways to tackle it.

The intuitive approach, which is what most C/C++ programmers think of, is that pointers are memory addresses. litb's example takes this approach. If you have a null pointer (which on most machines corresponds to the address 0), and you add the size of an int, you get the address 4. This implies that pointers are basically just fancy integers.

Unfortunately, there are a few problems with this. To begin with, it may not work. A null pointer is not guaranteed to actually use the address 0. (Although assigning the constant 0 to a pointer yields the null pointer).

Further, you're not allowed to increment the null pointer, or more generally, a pointer must always point to allocated memory (or one element past), or the special null pointer constant 0.

So a more correct way of thinking of it is that pointers are simply iterators allowing you to iterate over allocated memory. This is really one of the key ideas behind the STL iterators. They're modelled to behave very much as pointers, and to provide specializations that patch up raw pointers to work as proper iterators.

A more elaborate explanation of this is given here, for example.

But this latter view means that you should really explain STL iterators, and then simply say that pointers are a special case of these. You can increment a pointer to point to the next element in the buffer, just like you can a `std::vector<int>::iterator`. It can point one element past the end of an array, just like the end iterator in any other container. You can subtract two pointers *that point into the same buffer* to get the number of elements between them, just like you can with iterators, and just like with iterators, if the pointers point into separate buffers, you can *not* meaningfully compare them. (For a practical example of why not, consider what happens in a segmented memory space. What's the distance between two pointers pointing to separate segments?)

Of course in practice, there's a very close correlation between CPU addresses and C/C++ pointers. But they're not *exactly* the same thing. Pointers have a few limitations that may not be strictly necessary on your CPU.

Of course, most C++ programmers muddle by on the first understanding, even though it's technically incorrect. It's typically close enough to how your code ends up behaving that people think they get it, and move on.

But for someone coming from Java, and just learning about pointers from scratch, the latter explanation may be just as easily understood, and it's going to spring fewer surprises on them later.

answered Dec 27 '08 at 10:07

jalf
175k 35 244 479

It is important to be aware that a pointer is not always 4 bytes long. On some systems (64bit) it may be 8 bytes long. For example never assume a pointer is the same size as a integer. Too many errors have been made because programmers hide pointers in integers! – Waxhead Dec 27 '13 at 1:44

True, that's an important point. I intended the 4 byte thing to just be an example, but I should probably have made that clearer, because pointers are certainly no *guaranteed* to be 4 bytes, and they often aren't. – jalf Dec 27 '13 at 12:37

This is one pretty good at link here about Pointer Arithmetic

For example:

Pointer and array

Formula for computing the address of ptr + i where ptr has type T *. then the formula for the address is:

addr(ptr + i) = addr(ptr) + [sizeof(T) * i]

For for type of int on 32bit platform, addr(ptr+i) = addr(ptr)+4*i;

Subtraction

We can also compute ptr - i. For example, suppose we have an int array called arr. int arr[10] ; int * p1, * p2 ;

```
p1 = arr + 3 ; // p1 == & arr[ 3 ]
p2 = p1 - 2 ; // p1 == & arr[ 1 ]
```

answered Nov 4 '12 at 23:01

Gob00st
2,585 2 21 53

This may be true -- there is no guarantee that an int is 32 bits wide on a 32 bit platform -- the rule is that it must be able to represent [−32767,+32767], which just means it must be at least 16 bits wide. You might have a 32 bit platform where the compiler sees ints as 16 or 64 bit wide. – Clearer Feb 14 '15 at 0:04