2022 Developer Survey is open! Take survey. ×

With arrays, why is it the case that a[5] == 5[a]?

Asked 13 years, 4 months ago Modified 7 months ago Viewed 115k times

1747

As Joel points out in Stack Overflow podcast #34, in C Programming Language (aka: K & R), there is mention of this property of arrays in C: `a[5] == 5[a]`

Joel says that it's because of pointer arithmetic but I still don't understand. **Why does `a[5] == 5[a]`?**

c arrays pointers pointer-arithmetic

Share Edit Follow Flag — edited Oct 21, 2018 at 7:55 Cœur 34.6k ●24 ●185 ●250 — asked Dec 19, 2008 at 17:01 Dinah 50.8k ●30 ●130 ●149

- 56 — would something like a[+] also work like *(a++) OR *(++a) ? – Egon May 13, 2010 at 16:14
- 52 — @Egon: That's very creative but unfortunately that's not how compilers work. The compiler interprets `a[1]` as a series of tokens, not strings: *({integer location of}a {operator}+ {integer}1) is the same as *({integer}1 {operator}+ {integer location of}a) but is not the same as *({integer location of}a {operator}+ {operator}+) – Dinah May 13, 2010 at 17:24
- 14 — An interesting compound variation on this is illustrated in Illogical array access, where you have `char bar[]; int foo[];` and `foo[i][bar]` is used as an expression. – Jonathan Leffler Oct 17, 2012 at 6:38
- 6 — @EldritchConundrum, why do you think 'the compiler cannot check that the left part is a pointer'? Yes, it can. It's true that `a[b]` = `*(a + b)` for any given `a` and `b`, but it was the language designers' free choice for `+` to be defined commutative for all types. Nothing could prevent them from forbidding `i + p` while allowing `p + i`. – ach Mar 14, 2014 at 19:46
- 14 — @Andrey One usually expects `+` to be commutative, so maybe the real problem is choosing to make pointer operations resemble arithmetic, instead of designing a separate offset operator. – Eldritch Conundrum Mar 18, 2014 at 10:36

19 Answers

Sorted by: Highest score (default)

2060

The C standard defines the `[]` operator as follows:

`a[b] == *(a + b)`

Therefore `a[5]` will evaluate to:

```
*(a + 5)
```

and `5[a]` will evaluate to:

```
*(5 + a)
```

`a` is a pointer to the first element of the array. `a[5]` is the value that's 5 **elements** further from `a`, which is the same as `*(a + 5)`, and from elementary school math we know those are equal (addition is commutative).

Share Edit Follow Flag — edited Nov 30, 2016 at 10:47 Marco Bonelli 55.5k ●20 ●106 ●114 — answered Dec 19, 2008 at 17:04 mmx 402k ●87 ●836 ●780

- 351 — I wonder if it isn't more like *((5 * sizeof(a)) + a). Great explanation though. – John MacIntyre Dec 19, 2008 at 17:06
- 110 — @Dinah: From a C-compiler perspective, you are right. No sizeof is needed and those expressions I mentioned are THE SAME. However, the compiler will take sizeof into account when producing machine code. If a is an int array, `a[5]` will compile to something like `mov eax, [ebx+20]` instead of `[ebx+5]` – mmx Dec 19, 2008 at 17:18
- 13 — @Dinah: A is an address, say 0x1230. If a was in 32-bit int array, then a[0] is at 0x1230, a[1] is at 0x1234, a[2] at 0x1238...a[5] at x1244 etc. If we just add 5 to 0x1230, we get 0x1235, which is wrong. – James Curran Dec 19, 2008 at 17:21
- 43 — @sr105: That's a special case for the + operator, where one of the operands is a pointer and the other an integer. The standard says that the result will be of the type of the pointer. The compiler /has to be/ smart enough. – aib Dec 23, 2008 at 2:08
- 55 — "from elementary school math we know those are equal" - I understand that you are simplifying, but I'm with those who feel like this is oversimplifying. It's not elementary that `*(10 + (int *)13) != *((int *)10 + 13)`. In other words, there's more going on here than elementary school arithmetic. The commutativity relies critically on the compiler recognizing which operand is a pointer (and to what size of object). To put it another way, `(1 apple + 2 oranges) = (2 oranges + 1 apple)`, but `(1 apple + 2 oranges) != (1 orange + 2 apples)`. – LarsH Dec 1, 2010 at 20:54

301

Because array access is defined in terms of pointers. `a[i]` is defined to mean `*(a + i)`, which is commutative.

Share Edit Follow Flag — edited May 13, 2011 at 13:47 — answered Dec 19, 2008 at 17:05 David Thornley 55.4k ●8 ●89 ●155

- 51 — Arrays are not defined in terms of pointers, but access to them is. – Lightness Races in Orbit May 12, 2011 at 23:20
- 8 — I would add "so it is equal to `*(i + a)`, which can be written as `i[a]`". – Jim Balter Apr 5, 2013 at 22:11
- 4 — I would suggest you include the quote from the standard, which is as follows: 6.5.2.1: 2 A postfix expression followed by an expression in square brackets [] is a subscripted designation of an element of an array object. The definition of the subscript operator [] is that E1[E2] is identical to (*((E1)+(E2))). Because of the conversion rules that apply to the binary + operator, if E1 is an array object (equivalently, a pointer to the initial element of an array object) and E2 is an integer, E1[E2] designates the E2-th element of E1 (counting from zero). – Vality Feb 17, 2015 at 21:41
- 2 — Nitpick: It doesn't make sense to say that "`*(a + i)` is commutative". However, `*(a + i) = *(i + a) = i[a]` because addition is commutative. – Andreas Rejbrand Oct 13, 2019 at 22:18
- 1 — @AndreasRejbrand OTOH `+` is the only binary operator in the expression, so it's rather clear what can be commutative at all. – U. Windl Nov 4, 2020 at 13:03

264

I think something is being missed by the other answers.

Yes, `p[i]` is by definition equivalent to `*(p+i)`, which (because addition is commutative) is equivalent to `*(i+p)`, which (again, by the definition of the `[]` operator) is equivalent to `i[p]`.

(And in `array[i]`, the array name is implicitly converted to a pointer to the array's first element.)

But the commutativity of addition is not all that obvious in this case.

When both operands are of the same type, or even of different numeric types that are promoted to a common type, commutativity makes perfect sense: `x + y == y + x`.

But in this case we're talking specifically about pointer arithmetic, where one operand is a pointer and the other is an integer. (Integer + integer is a different operation, and pointer + pointer is nonsense.)

The C standard's description of the `+` operator (N1570 6.5.6) says:

> For addition, either both operands shall have arithmetic type, or one operand shall be a pointer to a complete object type and the other shall have integer type.

It could just as easily have said:

> For addition, either both operands shall have arithmetic type, or *the left* operand shall be a pointer to a complete object type and *the right operand* shall have integer type.

in which case both `i + p` and `i[p]` would be illegal.

In C++ terms, we really have two sets of overloaded `+` operators, which can be loosely described as:

```
pointer operator+(pointer p, integer i);
```

and

```
pointer operator+(integer i, pointer p);
```

of which only the first is really necessary.

So why is it this way?

C++ inherited this definition from C, which got it from B (the commutativity of array indexing is explicitly mentioned in the 1972 Users' Reference to B), which got it from BCPL (manual dated 1967), which may well have gotten it from even earlier languages (CPL? Algol?).

So the idea that array indexing is defined in terms of addition, and that addition, even of a pointer and an integer, is commutative, goes back many decades, to C's ancestor languages.

Those languages were much less strongly typed than modern C is. In particular, the distinction between pointers and integers was often ignored. (Early C programmers sometimes used pointers as unsigned integers, before the `unsigned` keyword was added to the language.) So the idea of making addition non-commutative because the operands are of different types probably wouldn't have occurred to the designers of those languages. If a user wanted to add two "things", whether those "things" are integers, pointers, or something else, it wasn't up to the language to prevent it.

And over the years, any change to that rule would have broken existing code (though the 1989 ANSI C standard might have been a good opportunity).

Changing C and/or C++ to require putting the pointer on the left and the integer on the right might break some existing code, but there would be no loss of real expressive power.

So now we have `arr[3]` and `3[arr]` meaning exactly the same thing, though the latter form should never appear outside the IOCCC.

Share Edit Follow Flag — edited Jan 28, 2016 at 20:48 — answered Aug 23, 2013 at 1:37 Keith Thompson 242k ●41 ●401 ●601

- 14 — Fantastic description of this property. From a high level view, I think `3[arr]` is an interesting artifact but should rarely if ever be used. The accepted answer to this question (<stackoverflow.com/q/1390365/356>) which I asked a while back has changed the way I've thought about syntax. Although there's often technically not a right and wrong way to do these things, these kinds of features start you thinking in a way which is separate from the implementation details. There's benefit to this different way of thinking which is in part lost when you fixate on the implementation details. – Dinah Aug 24, 2013 at 1:01
- 3 — Addition is commutative. For the C standard to define it otherwise would be strange. That's why it could not just as easily said "For addition, either both operands shall have arithmetic type, or the left operand shall be a pointer to a complete object type and the right operand shall have integer type." - That wouldn't make sense to most people who add things. – iheanyi Apr 21, 2014 at 17:54
- 11 — @iheanyi: Addition is usually commutative -- and it usually takes two operands of the same type. Pointer addition lets you add a pointer and an integer, but not two pointers. IMHO that's already a sufficiently odd special case that requiring the pointer to be the left operand wouldn't be a significant burden. (Some languages use "+" for string concatenation; that's certainly not commutative.) – Keith Thompson Apr 21, 2014 at 18:13
- 3 — @supercat, That's even worse. That would mean that sometimes x + 1 != 1 + x. That would completely violate the associative property of addition. – iheanyi Oct 21, 2014 at 16:34
- 4 — @iheanyi: I think you meant commutative property; addition is already not associative, since on most implementations (1LL+1U)-2 != 1LL+(1U-2). Indeed, the change would make some situations associative which presently aren't, e.g. 3U+(UINT_MAX-2L) would equal (3U+UINT_MAX)-2. What would be best, though, is for the language to have add new distinct types for promotable integers and "wrapping" algebraic rings, so that adding 2 to a `ring16_t` which holds 65535 would yield a `ring16_t` with value 1, *independent of the size of* `int`. – supercat Oct 21, 2014 at 16:46

206

And, of course

```
("ABCD"[2] == 2["ABCD"]) && (2["ABCD"] == 'C') && ("ABCD"[2] == 'C')
```

The main reason for this was that back in the 70's when C was designed, computers didn't have much memory (64KB was a lot), so the C compiler didn't do much syntax checking. Hence "`X[Y]`" was rather blindly translated into "`*(X+Y)`"

This also explains the "`+=`" and "`++`" syntaxes. Everything in the form "`A = B + C`" had the same compiled form. But, if B was the same object as A, then an assembly level optimization was available. But the compiler wasn't bright enough to recognize it, so the developer had to (`A += C`). Similarly, if `C` was `1`, a different assembly level optimization was available, and again the developer had to make it explicit, because the compiler didn't recognize it. (More recently compilers do, so those syntaxes are largely unnecessary these days)

Share Edit Follow Flag — edited Jul 28, 2017 at 17:51 franji1 2,805 ●2 ●21 ●41 — answered Dec 19, 2008 at 17:07 James Curran 98.4k ●35 ●176 ●255

- 132 — Actually, that evaluates to false; the first term "ABCD"[2] == 2["ABCD"] evaluates to true, or 1, and 1 != 'C' :D – Jonathan Leffler Dec 19, 2008 at 17:16
- 9 — @Jonathan: same ambiguity lead to the editing of the original title of this post. Are we the equal marks mathematical equivalency, code syntax, or pseudo-code. I argue mathematical equivalency but since we're talking about code, we can't escape that we're viewing everything in terms of code syntax. – Dinah Dec 19, 2008 at 17:26
- 20 — Isn't this a myth? I mean that the += and ++ operators were created to simplify for the compiler? Some code gets clearer with them, and it is useful syntax to have, no matter what the compiler does with it. – Thomas Padron-McCarthy Dec 19, 2008 at 17:44
- 6 — += and ++ has another significant benefit. if the left hand side changes some variable while evaluated, the change will only done once. a = a + ...; will do it twice. – Johannes Schaub - litb Dec 19, 2008 at 17:49
- 9 — No - "ABCD"[2] == *("ABCD" + 2) = *("CD") = 'C'. Dereferencing a string gives you a char, not a substring – MSalters Sep 21, 2009 at 10:34

58

One thing no-one seems to have mentioned about Dinah's problem with `sizeof`:

You can only add an integer to a pointer, you can't add two pointers together. That way when adding a pointer to an integer, or an integer to a pointer, the compiler always knows which bit has a size that needs to be taken into account.

Share Edit Follow Flag — edited Dec 18, 2009 at 8:01 Dinah 50.8k ●30 ●130 ●149 — answered Feb 11, 2009 at 15:56 user30364 614 ●4 ●2

- 2 — There's a fairly exhaustive conversation about this in the comments of the accepted answer. I referenced said conversation in the edit to the original question but did not directly address your very valid concern of sizeof. Not sure how to best do this in SO. Should I make another edit to the orig. question? – Dinah Apr 21, 2009 at 13:51
- I'd like to note that you cannot *add* pointers, but you can *subtract* pointers (returning the number of items between). – U. Windl Nov 4, 2020 at 13:10

51

To answer the question literally. It is not always true that `x == x`

```
double zero = 0.0;
double a[] = { 0,0,0,0,0, zero/zero}; // NaN
cout << (a[5] == 5[a] ? "true" : "false") << endl;
```

prints

```
false
```

Share Edit Follow Flag — answered Aug 11, 2011 at 13:50 Peter Lawrey 513k ●74 ●729 ●1106

- 22 — Actually a "nan" is not equal to itself: `cout << (a[5] == a[5] ? "true" : "false") << endl;` is `false`. – TrueY Apr 23, 2013 at 9:34
- 10 — @TrueY: He did state that specifically for the NaN case (and specifically that `x == x` is not always true). I think that was his intention. So he is *technically* correct (and possibly, as they say, the best kind of correct!). – Tim Čas Feb 13, 2015 at 1:04
- 6 — The question is about C, your code is not C code. There is also a `NAN` in `<math.h>`, which is better than `0.0/0.0`, because `0.0/0.0` is UB when `__STDC_IEC_559__` is not defined (Most implementations do not define `__STDC_IEC_559__`, but on most implementations `0.0/0.0` will still work) – 12431234123412341234123 May 14, 2018 at 16:02

32

I just find out this ugly syntax could be "useful", or at least very fun to play with when you want to deal with an array of indexes which refer to positions into the same array. It can replace nested square brackets and make the code more readable !

```
int a[] = { 2 , 3 , 3 , 2 , 4 };
int s = sizeof a / sizeof *a;  //  s == 5

for(int i = 0 ; i < s ; ++i) {

    cout << a[a[a[i]]] << endl;
    // ... is equivalent to ...
    cout << i[a][a][a] << endl;  // but I prefer this one, it's easier to increase the level of indirection (without loop)

}
```

Of course, I'm quite sure that there is no use case for that in real code, but I found it interesting anyway :)

Share Edit Follow Flag — edited Aug 31, 2020 at 3:03 Good Night Nerd Pride 6,969 ●4 ●46 ●59 — answered Jun 19, 2012 at 19:50 Frédéric Terrazzoni 2,070 ●18 ●25

- 1 — When you see `i[a][a][a]` you think i is either a pointer to an array or an array of a pointer to an array or a array ... and `a` is a index. When you see `a[a[a[i]]]`, you think a is a pointer to a array or a array and `i` is a index. – 12431234123412341234123 May 14, 2018 at 11:58
- 3 — Wow! It's very cool usage of this "stupid" feature. Could be useful in algorithmic contest in some problems)) – Serge Breusov Jun 26, 2018 at 8:53
- The question is about C, your code is not C code. – 12431234123412341234123 Aug 11, 2021 at 15:15

28

Nice question/answers.

Just want to point out that C pointers and arrays are not the *same*, although in this case the difference is not essential.

Consider the following declarations:

```
int a[10];
int* p = a;
```

In `a.out`, the symbol `a` is at an address that's the beginning of the array, and symbol `p` is at an address where a pointer is stored, and the value of the pointer at that memory location is the beginning of the array.

Share Edit Follow Flag — edited Jun 1, 2020 at 4:54 NAND 655 ●8 ●22 — answered Dec 20, 2008 at 8:16 PolyThinker 5,076 ●20 ●22

- 2 — No, technically they are not the same. If you define some b as int*const and make it point to an array, it is still a pointer, meaning that in the symbol table, b refers to a memory location that stores an address, which in turn points to where the array is. – PolyThinker Dec 22, 2008 at 5:42
- 4 — Very good point. I remember having a very nasty bug when I defined a global symbol as char s[100] in one module, declare it as extern char *s; in another module. After linking it all together the program behaved very strangely. Because the module using the extern declaration was using the initial bytes of the array as a pointer to char. – Giorgio May 2, 2012 at 18:15
- 1 — Originally, in C's grandparent BCPL, an array was a pointer. That is, what you got when you wrote (I have transliterated to C) `int a[10]` was a pointer called 'a', which pointed to enough store for 10 integers, elsewhere. Thus a+i and j+i had the same form: add the contents of a couple of memory locations. In fact, I think BCPL was typeless, so they were identical. And the sizeof-type scaling did not apply, since BCPL was purely word-oriented (on word-addressed machines also). – dave May 3, 2012 at 2:33
- I think the best way to understand the difference is to compare `int*p = a;` to `int b = 5;` In the latter, "b" and "5" are both integers, but "b" is a variable, while "5" is a fixed value. Similarly, "p" & "a" are both addresses of a character, but "a" is a fixed value. – James Curran Mar 12, 2013 at 16:34
- While this "answer" does not answer the question (and thus should be a comment, not an answer), you could summarize as "an array is not an lvalue, but a pointer is". – U. Windl Nov 4, 2020 at 13:15

21

For pointers in C, we have

```
a[5] == *(a + 5)
```

and also

```
5[a] == *(5 + a)
```

Hence it is true that `a[5] == 5[a].`

Share Edit Follow Flag — edited Mar 23, 2012 at 17:36 user1055604 1,582 ●10 ●28 — answered Mar 23, 2012 at 7:05 user1287577

18

Not an answer, but just some food for thought. If class is having overloaded index/subscript operator, the expression `0[x]` will not work:

```
class Sub
{
public:
    int operator [](size_t nIndex)
    {
        return 0;
    }
};

int main()
{
    Sub s;
    s[0];
    0[s]; // ERROR
}
```

Since we dont have access to **int** class, this cannot be done:

```
class int
{
   int operator[](const Sub&);
};
```

Share Edit Follow Flag — edited Dec 9, 2015 at 18:29 — answered Jun 19, 2011 at 8:37 Ajay 17.4k ●10 ●53 ●96

- 3 — `class Sub { public:     int operator[](size_t nIndex) const     {         return 0;     }         friend int operator[](size_t nIndex, const Sub& This)     {         return 0;     }   };` – Ben Voigt Apr 5, 2013 at 17:23
- 1 — Have you actually tried compiling it? There are set of operators that cannot be implemented outside class (i.e. as non-static functions)! – Ajay Apr 5, 2013 at 21:10
- 4 — oops, you're right. "`operator[]` shall be a non-static member function with exactly one parameter." I was familiar with that restriction on `operator=`, didn't think it applied to `[]`. – Ben Voigt Apr 5, 2013 at 21:21
- 2 — Of course, if you change the definition of `[]` operator, it would never be equivalent again... if `a[b]` is equal to `*(a + b)` and you change this, you'll have to overload also `int::operator[](const Sub&);` and `int` is not a class... – Luis Colorado Sep 19, 2014 at 13:18
- 10 — This...isn't...C. – MD XF Dec 13, 2016 at 7:13

14

It has very good explanation in *A TUTORIAL ON POINTERS AND ARRAYS IN C* by Ted Jensen.

Ted Jensen explained it as:

> In fact, this is true, i.e wherever one writes `a[i]` it can be replaced with `*(a + i)` without any problems. In fact, the compiler will create the same code in either case. Thus we see that pointer arithmetic is the same thing as array indexing. Either syntax produces the same result.
>
> This is NOT saying that pointers and arrays are the same thing, they are not. We are only saying that to identify a given element of an array we have the choice of two syntaxes, one using array indexing and the other using pointer arithmetic, which yield identical results.
>
> Now, looking at this last expression, part of it.. `(a + i)`, is a simple addition using the + operator and the rules of C state that such an expression is commutative. That is (a + i) is identical to `(i + a)`. Thus we could write `*(i + a)` just as easily as `*(a + i)`. But `*(i + a)` could have come from `i[a]` ! From all of this comes the curious truth that if:
>
> ```
> char a[20];
> ```
>
> writing
>
> ```
> a[3] = 'x';
> ```
>
> is the same as writing
>
> ```
> 3[a] = 'x';
> ```

Share Edit Follow Flag — edited Jan 13, 2017 at 6:00 Right leg 14.8k ●6 ●44 ●74 — answered Sep 27, 2013 at 6:46 A.s. Bhullar 2,510 ●2 ●23 ●32

- 4 — a+i is NOT simple addition, because it's pointer arithmetic. if the size of the element of a is 1 (char), then yes, it's just like integer +. But if it's (e.g.) an integer, then it might be equivalent to a + 4*i. – Alex Brown Dec 4, 2015 at 20:17
- @AlexBrown Yes, it is pointer arithmetic, which is exactly why your last sentence is wrong, unless you first cast 'a' to be a (char*) (assuming that an int is 4 chars). I really don't understand why so many people are getting hung up on the actual value result of pointer arithmetic. Pointer arithmetic's entire purpose is to abstract away the underlying pointer values and let the programmer think about the objects being manipulated rather than address values. – jschultz410 Mar 21, 2018 at 16:11

9

I know the question is answered, but I couldn't resist sharing this explanation.

I remember Principles of Compiler design, Let's assume `a` is an `int` array and size of `int` is 2 bytes, & Base address for `a` is 1000.

How `a[5]` will work ->

```
Base Address of your Array a + (5*size of(data type for array a))
i.e. 1000 + (5*2) = 1010
```

So,

Similarly when the c code is broken down into 3-address code, `5[a]` will become ->

```
Base Address of your Array a + (size of(data type for array a)*5)
i.e. 1000 + (2*5) = 1010
```

So basically both the statements are pointing to the same location in memory and hence, `a[5] = 5[a]`.

This explanation is also the reason why negative indexes in arrays work in C.

i.e. if I access `a[-5]` it will give me

```
Base Address of your Array a + (-5 * size of(data type for array a))
i.e. 1000 + (-5*2) = 990
```

It will return me object at location 990.

Share Edit Follow Flag — edited Nov 29, 2016 at 1:18 — answered May 4, 2016 at 8:24 Ajinkya Patil 721 ●1 ●6 ●16

8

In C arrays, `arr[3]` and `3[arr]` are the same, and their equivalent pointer notations are `*(arr + 3)` to `*(3 + arr)`. But on the contrary `[arr]3` or `[3]arr` is not correct and will result into syntax error, as `(arr + 3)*` and `(3 + arr)*` are not valid expressions. The reason is dereference operator should be placed before the address yielded by the expression, not after the address.

Share Edit Follow Flag — edited Jan 11, 2021 at 19:31 Machavity ♦ 29.8k ●26 ●86 ●96 — answered Dec 17, 2013 at 11:22 Krishan 277 ●4 ●7

8

in c compiler

```
a[i]
i[a]
*(a+i)
```

are different ways to refer to an element in an array ! (NOT AT ALL WEIRD)

Share Edit Follow Flag — answered Oct 29, 2014 at 9:14 AVIK DUTTA 724 ●5 ●21

7

A little bit of history now. Among other languages, BCPL had a fairly major influence on C's early development. If you declared an array in BCPL with something like:

```
let V = vec 10
```

that actually allocated 11 words of memory, not 10. Typically V was the first, and contained the address of the immediately following word. So unlike C, naming V went to that location and picked up the address of the zeroeth element of the array. Therefore array indirection in BCPL, expressed as

```
let J = V!5
```

really did have to do `J = !(V + 5)` (using BCPL syntax) since it was necessary to fetch V to get the base address of the array. Thus `V!5` and `5!V` were synonymous. As an anecdotal observation, WAFL (Warwick Functional Language) was written in BCPL, and to the best of my memory tended to use the latter syntax rather than the former for accessing the nodes used as data storage. Granted this is from somewhere between 35 and 40 years ago, so my memory is a little rusty. :)

The innovation of dispensing with the extra word of storage and having the compiler insert the base address of the array when it was named came later. According to the C history paper this happened at about the time structures were added to C.

Note that `!` in BCPL was both a unary prefix operator and a binary infix operator, in both cases doing indirection. just that the binary form included an addition of the two operands before doing the indirection. Given the word oriented nature of BCPL (and B) this actually made a lot of sense. The restriction of "pointer and integer" was made necessary in C when it gained data types, and `sizeof` became a thing.

Share Edit Follow Flag — answered Apr 8, 2019 at 18:45 dgnuff 2,801 ●2 ●17 ●28

2

In C

```
int a[]={10,20,30,40,50};
int *p=a;
printf("%d\n",*p++);//output will be 10
printf("%d\n",*a++);//will give an error
```

Pointer `p` is a "variable", array name `a` is a "mnemonic" or "synonym", so `p++` is valid but `a++` is invalid.

`a[2]` is equals to `2[a]` because the internal operation on both of this is "Pointer Arithmetic" internally calculated as `*(a+2)` equals `*(2+a)`

Share Edit Follow Flag — edited Nov 7, 2020 at 3:32 U. Windl 2,749 ●20 ●47 — answered Feb 12, 2017 at 13:54 Jayghosh Wankar 309 ●2 ●8

2

Well, this is a feature that is only possible because of the language support.

The compiler interprets `a[i]` as `*(a+i)` and the expression `5[a]` evaluates to `*(5+a)`. Since addition is commutative it turns out that both are equal. Hence the expression evaluates to `true`.

Share Edit Follow Flag — answered Apr 2, 2018 at 18:42 Harsha J K 141 ●1 ●15

- Although redundant this is clear, concise and short. – Bill K Dec 5, 2019 at 19:04

2

Because C compiler always convert array notation in pointer notation. `a[5] = *(a + 5)` also `5[a] = *(5 + a) = *(a + 5)` So, both are equal.

Share Edit Follow Flag — answered Aug 31, 2021 at 9:11 bug_29 318 ●2 ●13

1

Because it's useful to avoid confusing nesting.

Would you rather read this:

```
array[array[head].next].prev
```

or this:

```
head[array].next[array].prev
```

Incidentally, C++ has a similar commutative property for function calls. Rather than writing `g(f(x))` as you must in C, you may use member functions to write `x.f().g()`. Replace f and g with lookup tables and you can write `g[f[x]]` (functional style) or `(x[f])[g]` (oop style). The latter gets really nice with structs containing indices: `x[xs].y[ys].z[zs]`. Using the more common notation that's `zs[ys[xs[x].y].z]`.

Share Edit Follow Flag — edited Sep 25, 2021 at 18:18 — answered Sep 20, 2021 at 11:39 Samuel Danielson 4,813 ●3 ●34 ●33