Is an array name a pointer?

Asked 12 years, 6 months ago Modified 1 year, 8 months ago Viewed 92k times

243

Is an array's name a pointer in C? If not, what is the difference between an array's name and a pointer variable?

c arrays pointers

175

Share Edit Follow Flag — edited May 22, 2017 at 8:30 Lundin 174k ●38 ●234 ●367 — asked Oct 29, 2009 at 6:38 user188276

5 No. But *array* is the same *&array[0]* – user166390 Oct 29, 2009 at 6:51

37 @pst: `&array[0]` yields a pointer, not an array ;) – jalf Oct 29, 2009 at 6:55

@pst meant that array (the name of array, that's why in italic) points on the first element of array and it's the same as write &array[0] (also in italic). – Nava Carmon Oct 29, 2009 at 7:23

32 @Nava (and pst): *array* and *&array[0]* are not really the same. Case in point: *sizeof(array)* and *sizeof(&array[0])* give different results. – Thomas Padron-McCarthy Oct 29, 2009 at 7:50

1 @Thomas agree, but in terms of pointers, when you dereference array and &array[0], they produce the same value of array[0].i.e. *array == array[0]. Nobody meant that these two pointers are the same, but in this specific case (pointing to the first element) you can use the name of array either. – Nava Carmon Oct 29, 2009 at 11:12

You might look at this question: stackoverflow.com/questions/1054247/... – Andrew Y Oct 29, 2009 at 14:57

1 These might also help in your understanding: stackoverflow.com/questions/381542 , stackoverflow.com/questions/660752 – Dinah Oct 29, 2009 at 15:06

9 Answers

Sorted by: Highest score (default)

297

An array is an array and a pointer is a pointer, but in most cases array names are *converted* to pointers. A term often used is that they *decay* to pointers.

Here is an array:

```
int a[7];
```

`a` contains space for seven integers, and you can put a value in one of them with an assignment, like this:

```
a[3] = 9;
```

Here is a pointer:

```
int *p;
```

`p` doesn't contain any spaces for integers, but it can point to a space for an integer. We can, for example, set it to point to one of the places in the array `a`, such as the first one:

```
p = &a[0];
```

What can be confusing is that you can also write this:

```
p = a;
```

This does *not* copy the contents of the array `a` into the pointer `p` (whatever that would mean). Instead, the array name `a` is converted to a pointer to its first element. So that assignment does the same as the previous one.

Now you can use `p` in a similar way to an array:

```
p[3] = 17;
```

The reason that this works is that the array dereferencing operator in C, `[ ]`, is defined in terms of pointers. `x[y]` means: start with the pointer `x`, step `y` elements forward after what the pointer points to, and then take whatever is there. Using pointer arithmetic syntax, `x[y]` can also be written as `*(x+y)`.

For this to work with a normal array, such as our `a`, the name `a` in `a[3]` must first be converted to a pointer (to the first element in `a`). Then we step 3 elements forward, and take whatever is there. In other words: take the element at position 3 in the array. (Which is the fourth element in the array, since the first one is numbered 0.)

So, in summary, array names in a C program are (in most cases) converted to pointers. One exception is when we use the `sizeof` operator on an array. If `a` was converted to a pointer in this context, `sizeof a` would give the size of a pointer and not of the actual array, which would be rather useless, so in that case `a` means the array itself.

Share Edit Follow Flag — edited Oct 23, 2018 at 19:05 Deduplicator 43.3k ●6 ●62 ●109 — answered Oct 29, 2009 at 6:39 Thomas Padron-McCarthy 26.4k ●8 ●49 ●74

7 A similar automatic conversion is applied to function pointers - both `functionpointer()` and `(*functionpointer)()` mean the same thing, strangely enough. – Carl Norum Oct 29, 2009 at 6:52

3 He did not asked if arrays and pointers are the same, but if an array's name is a pointer – Ricardo Amores Oct 29, 2009 at 6:58

36 An array name is not a pointer. It's an identifier for a variable of type array, which has an implicit conversion to pointer of element type. – Pavel Minaev Oct 29, 2009 at 7:24

31 Also, apart from `sizeof()`, the other context in which there's no array->pointer decay is operator `&` - in your example above, `&a` will be a pointer to an array of 7 `int`, not a pointer to a single `int`; that is, its type will be `int(*)[7]`, which is not implicitly convertible to `int*`. This way, functions can actually take pointers to arrays of specific size, and enforce the restriction via the type system. – Pavel Minaev Oct 29, 2009 at 7:25

4 @onmyway133, check here for a short explanation and further citations. – Carl Norum Feb 12, 2015 at 15:39

45

When an array is used as a value, its name represents the address of the first element.
When an array is not used as a value its name represents the whole array.

```
int arr[7];

/* arr used as value */
foo(arr);
int x = *(arr + 1); /* same as arr[1] */

/* arr not used as value */
size_t bytes = sizeof arr;
void *q = &arr; /* void pointers are compatible with pointers to any object */
```

Share Edit Follow Flag — edited Jun 4, 2013 at 20:24 — answered Oct 29, 2009 at 9:03 pmg 103k ●11 ●122 ●196

30

If an expression of array type (such as the array name) appears in a larger expression and it isn't the operand of either the `&` or `sizeof` operators, then the type of the array expression is converted from "N-element array of T" to "pointer to T", and the value of the expression is the address of the first element in the array.

In short, the array name is not a pointer, but in most contexts it is treated *as though* it were a pointer.

Edit

Answering the question in the comment:

> If I use sizeof, do i count the size of only the elements of the array? Then the array "head" also takes up space with the information about length and a pointer (and this means that it takes more space, than a normal pointer would)?

When you create an array, the only space that's allocated is the space for the elements themselves; no storage is materialized for a separate pointer or any metadata. Given

```
char a[10];
```

what you get in memory is

```
   +---+
a: |   | a[0]
   +---+
   |   | a[1]
   +---+
   |   | a[2]
   +---+
    ...
   +---+
   |   | a[9]
   +---+
```

The *expression* `a` refers to the entire array, but there's no *object* `a` separate from the array elements themselves. Thus, `sizeof a` gives you the size (in bytes) of the entire array. The expression `&a` gives you the address of the array, *which is the same as the address of the first element*. The difference between `&a` and `&a[0]` is the type of the result[1] - `char (*)[10]` in the first case and `char *` in the second.

Where things get weird is when you want to access individual elements - the expression `a[i]` is defined as the result of `*(a + i)` - given an address value `a`, offset `i` elements (*not bytes*) from that address and dereference the result.

The problem is that `a` isn't a pointer or an address - it's the entire array object. Thus, the rule in C that whenever the compiler sees an expression of array type (such as `a`, which has type `char [10]`) *and* that expression isn't the operand of the `sizeof` or unary `&` operators, the type of that expression is converted ("decays") to a pointer type (`char *`), and the value of the expression is the address of the first element of the array. Therefore, the *expression* `a` has the same type and value as the expression `&a[0]` (and by extension, the expression `*a` has the same type and value as the expression `a[0]`).

C was derived from an earlier language called B, and in B `a` *was* a separate pointer object from the array elements `a[0]`, `a[1]`, etc. Ritchie wanted to keep B's array semantics, but he didn't want to mess with storing the separate pointer object. So he got rid of it. Instead, the compiler will convert array expressions to pointer expressions during translation as necessary.

Remember that I said arrays don't store any metadata about their size. As soon as that array expression "decays" to a pointer, all you have is a pointer to a single element. That element may be the first of a sequence of elements, or it may be a single object. There's no way to know based on the pointer itself.

When you pass an array expression to a function, all the function receives is a pointer to the first element - it has no idea how big the array is (this is why the `gets` function was such a menace and was eventually removed from the library). For the function to know how many elements the array has, you must either use a sentinel value (such as the 0 terminator in C strings) or you must pass the number of elements as a separate parameter.

1. Which *may* affect how the address value is interpreted - depends on the machine.

Share Edit Follow Flag — edited May 13, 2019 at 11:33 Greenberet 480 ●1 ●7 ●18 — answered Oct 29, 2009 at 14:54 John Bode 113k ●18 ●112 ●189

Have been looking for quite a long time for this answer. Thank you! And if you know, could you tell a little further what an array expression is. If I use sizeof, do i count the size of only the elements of the array? Then the array "head" also takes up space with the information about length and a pointer (and this means that it takes more space, than a normal pointer would)? – Andriy Dmytruk Dec 7, 2017 at 21:51

And one more thing. An array of length 5 is of type int[5]. So that is from where we know the length when we call sizeof(array) - from its type? And this means that arrays of different length are like different types of constants? – Andriy Dmytruk Dec 9, 2017 at 13:09

@AndriyDmytruk: `sizeof` is an operator, and it evaluates to the number *bytes* in the operand (either an expression denoting an object, or a type name in parentheses). So, for an array, `sizeof` evaluates to the number of elements multiplied by the number of bytes in a single element. If an `int` is 4 bytes wide, then a 5-element array of `int` takes up 20 bytes. – John Bode Dec 9, 2017 at 23:40

Isn't the operator `[ ]` special too? For example, `int a[2][3];`, then for `x = a[1][2];`, though it can be rewritten as `x = *( *(a+1) + 2 );`, here `a` isn't converted to a pointer type `int*` (though if `a` is an argument of a function it should be converted to `int*`). – Stan Jun 21, 2018 at 17:17

2 @Stan: The expression `a` has type `int [2][3]`, which "decays" to type `int (*)[3]`. The expression `*(a + 1)` has type `int [3]`, which "decays" to `int *`. Thus, `*(*(a + 1) + 2)` will have type `int`. `a` points to the first 3-element array of `int`, `a + 1` points to the second 3-element array of `int`, `*(a + 1)` *is* the second 3-element array of `int`, `*(a + 1) + 2` points to the third element of the second array of `int`, so `*(*(a + 1) + 2)` *is* the third element of the second array of `int`. How that gets mapped to machine code is entirely up to the compiler. – John Bode Jun 21, 2018 at 21:03

5

An array declared like this

```
int a[10];
```

allocates memory for 10 `int`s. You can't modify `a` but you can do pointer arithmetic with `a`.

A pointer like this allocates memory for just the pointer `p`:

```
int *p;
```

It doesn't allocate any `int`s. You can modify it:

```
p = a;
```

and use array subscripts as you can with a:

```
p[2] = 5;
a[2] = 5;    // same
*(p+2) = 5;  // same effect
*(a+2) = 5;  // same effect
```

Share Edit Follow Flag — edited Jun 3, 2011 at 16:37 — answered Oct 29, 2009 at 6:50 Grumdrig 15.8k ●14 ●54 ●68

2 Arrays are not always allocated on the stack. That's an implementation detail that will vary from compiler to compiler. In most cases static or global arrays will be allocated from a different memory region than the stack. Arrays of const types may be allocated from yet another region of memory – Mark Bessey Oct 29, 2009 at 6:59

1 I think Grumdrig meant to say "allocates 10 `int`s with automatic storage duration". – Lightness Races in Orbit May 30, 2011 at 21:01

4

The array name by itself yields a memory location, so you can treat the array name like a pointer:

```
int a[7];

a[0] = 1976;
a[1] = 1984;

printf("memory location of a: %p", a);

printf("value at memory location %p is %d", a, *a);
```

And other nifty stuff you can do to pointer (e.g. adding/substracting an offset), you can also do to an array:

```
printf("value at memory location %p is %d", a + 1, *(a + 1));
```

Language-wise, if C didn't expose the array as just **some sort of "pointer"**(pedantically it's just a memory location. It cannot point to arbitrary location in memory, nor can be controlled by the programmer). We always need to code this:

```
printf("value at memory location %p is %d", &a[1], a[1]);
```

Share Edit Follow Flag — edited Oct 29, 2009 at 15:03 Peter Mortensen 30k ●21 ●100 ●124 — answered Oct 29, 2009 at 7:29 Michael Buen 37.2k ●8 ●88 ●113

Doesn't this code cause UB when `sizeof (int*) != sizeof (void*)`? To be fair, i don't know any system where this is the case. – 12431234123412341234123 Jul 30, 2021 at 20:16

2

The following example provides a concrete difference between an array name and a pointer. Let say that you want to represent a 1D line with some given maximum dimension, you could do it either with an array or a pointer:

```
typedef struct {
   int length;
   int line_as_array[1000];
   int* line_as_pointer;
} Line;
```

Now let's look at the behavior of the following code:

```
void do_something_with_line(Line line) {
   line.line_as_pointer[0] = 0;
   line.line_as_array[0] = 0;
}

void main() {
   Line my_line;
   my_line.length = 20;
   my_line.line_as_pointer = (int*) calloc(my_line.length, sizeof(int));

   my_line.line_as_pointer[0] = 10;
   my_line.line_as_array[0] = 10;

   do_something_with_line(my_line);

   printf("%d %d\n", my_line.line_as_pointer[0], my_line.line_as_array[0]);
};
```

This code will output:

```
0 10
```

That is because in the function call to `do_something_with_line` the object was copied so:

1. The pointer `line_as_pointer` still contains the same address it was pointing to
2. The array `line_as_array` was copied to a new address which does not outlive the scope of the function

So while arrays are not given by values when you directly input them to functions, when you encapsulate them in structs they are given by value (i.e. copied) which outlines here a major difference in behavior compared to the implementation using pointers.

Share Edit Follow Flag — answered Sep 4, 2020 at 15:52 Cosmo Sterin 71 ●6

2

I think this example sheds some light on the issue:

```
#include <stdio.h>
int main()
{
        int a[3] = {9, 10, 11};
        int **b = &a;

        printf("a == &a: %d\n", a == b);
        return 0;
}
```

It compiles fine (with 2 warnings) in gcc 4.9.2, and prints the following:

```
a == &a: 1
```

oops :-)

So, the conclusion is no, the array is not a pointer, it is not stored in memory (not even read-only one) as a pointer, even though it looks like it is, since you can obtain its address with the & operator. But - oops - that operator does not work :-)), either way, you've been warned:

```
p.c: In function ‘main’:
pp.c:6:12: warning: initialization from incompatible pointer type
  int **b = &a;
            ^
p.c:8:28: warning: comparison of distinct pointer types lacks a cast
  printf("a == &a: %d\n", a == b);
```

C++ refuses any such attempts with errors in compile-time.

Edit:

This is what I meant to demonstrate:

```
#include <stdio.h>
int main()
{
    int a[3] = {9, 10, 11};
    void *c = a;

    void *b = &a;
    void *d = &c;

    printf("a == &a: %d\n", a == b);
    printf("c == &c: %d\n", c == d);
    return 0;
}
```

Even though `c` and `a` "point" to the same memory, you can obtain address of the `c` pointer, but you cannot obtain the address of the `a` pointer.

Share Edit Follow Flag — edited Apr 2, 2018 at 21:07 — answered Nov 8, 2015 at 14:53 Palo 912 ●12 ●25

2 "It compiles fine (with 2 warnings)". That's not fine. If you tell gcc to compile it as proper standard C by adding `-std=c11 -pedantic-errors`, you get a compiler error for writing invalid C code. The reason why is because you try to assign a `int (*)[3]` to a variable of `int**`, which are two types that have absolutely nothing to do with each other. So what this example is supposed to prove, I have no idea. – Lundin Mar 1, 2018 at 10:10

Thank you Lundin for your comment. You know there are many standards. I tried to clarify what I meant in the edit. The `int **` type is not the point there, one should better use the `void *` for this. – Palo Apr 2, 2018 at 21:09

-3

The array name behaves like a pointer and points to the first element of the array. Example:

```
int a[]={1,2,3};
printf("%p\n",a);     //result is similar to 0x7fff6fe40bc0
printf("%p\n",&a[0]); //result is similar to 0x7fff6fe40bc0
```

Both the print statements will give exactly same output for a machine. In my system it gave:

```
0x7fff6fe40bc0
```

Share Edit Follow Flag — answered Dec 1, 2014 at 17:59 Amitesh Ranjan 1,034 ●1 ●12 ●9

-5

Array name is the address of 1st element of an array. So yes array name is a const pointer.

Share Edit Follow Flag — answered Sep 28, 2015 at 11:14 SAQIB SOHAIL BHATTI 67 ●1 ●7