What is the difference between char array and char pointer in C?

Asked 9 years, 11 months ago Modified 1 year, 11 months ago Viewed 410k times

265

I am trying to understand pointers in C but I am currently confused with the following:

```
char *p = "hello"
```

This is a char pointer pointing at the character array, starting at h.

```
char p[] = "hello"
```

This is an array that stores hello.

What is the difference when I pass both these variables into this function?

```
void printSomething(char *p)
{
    printf("p: %s",p);
}
```

arrays pointers

Share Edit Follow Flag

This would not be valid: `char p[3] = "hello";` The initializer string is too long for the size of the array you declare. Typo?

Or just `char p[]="hello";` would suffice!

possible duplicate of C: differences between char pointer and array

8 Answers

262

`char*` and `char[]` *are different types*, but it's not immediately apparent in all cases. This is because arrays *decay into pointers*, meaning that if an expression of type `char[]` is provided where one of type `char*` is expected, the compiler automatically converts the array into a pointer to its first element.

Your example function `printSomething` expects a pointer, so if you try to pass an array to it like this:

```
char s[10] = "hello";
printSomething(s);
```

The compiler pretends that you wrote this:

```
char s[10] = "hello";
printSomething(&s[0]);
```

Share Edit Follow Flag

99

Let's see:

```
#include <stdio.h>
#include <string.h>

int main()
{
    char *p = "hello";
    char q[] = "hello"; // no need to count this

    printf("%zu\n", sizeof(p)); // => size of pointer to char -- 4 on x86, 8 on x86-64
    printf("%zu\n", sizeof(q)); // => size of char array in memory -- 6 on both

    // size_t strlen(const char* s) and we don't get any warnings here:
    printf("%zu\n", strlen(p)); // => 5
    printf("%zu\n", strlen(q)); // => 5

    return 0;
}
```

foo* and foo[] are different types and they are handled differently by the compiler (pointer = address + representation of the pointer's type, array = pointer + optional length of the array, if known, for example, if the array is statically allocated), the details can be found in the standard. And at the level of runtime no difference between them (in assembler, well, almost, see below).

Also, there is a related question in the C FAQ:

> **Q**: What is the difference between these initializations?
>
> ```
> char a[] = "string literal";
> char *p = "string literal";
> ```
>
> My program crashes if I try to assign a new value to p[i].
>
> **A**: A string literal (the formal term for a double-quoted string in C source) can be used in two slightly different ways:
>
> 1. As the initializer for an array of char, as in the declaration of char a[] , it specifies the initial values of the characters in that array (and, if necessary, its size).
> 2. Anywhere else, it turns into an unnamed, static array of characters, and this unnamed array may be stored in read-only memory, and which therefore cannot necessarily be modified. In an expression context, the array is converted at once to a pointer, as usual (see section 6), so the second declaration initializes p to point to the unnamed array's first element.
>
> Some compilers have a switch controlling whether string literals are writable or not (for compiling old code), and some may have options to cause string literals to be formally treated as arrays of const char (for better error catching).
>
> See also questions 1.31, 6.1, 6.2, 6.8, and 11.8b.
>
> References: K&R2 Sec. 5.5 p. 104
>
> ISO Sec. 6.1.4, Sec. 6.5.7
>
> Rationale Sec. 3.1.4
>
> H&S Sec. 2.7.4 pp. 31-2

Share Edit Follow Flag

51

> What is the difference between char array vs char pointer in C?

C99 N1256 draft

There are two different uses of character string literals:

1. Initialize `char[]`:

   ```
   char c[] = "abc";
   ```

   This is "more magic", and described at 6.7.8/14 "Initialization":

   > An array of character type may be initialized by a character string literal, optionally enclosed in braces. Successive characters of the character string literal (including the terminating null character if there is room or if the array is of unknown size) initialize the elements of the array.

   So this is just a shortcut for:

   ```
   char c[] = {'a', 'b', 'c', '\0'};
   ```

   Like any other regular array, `c` can be modified.

2. Everywhere else: it generates an:
   - unnamed
   - array of char What is the type of string literals in C and C++?
   - with static storage
   - that gives UB (undefined behavior) if modified

   So when you write:

   ```
   char *c = "abc";
   ```

   This is similar to:

   ```
   /* __unnamed is magic because modifying it gives UB. */
   static char __unnamed[] = "abc";
   char *c = __unnamed;
   ```

   Note the implicit cast from `char[]` to `char *`, which is always legal.

   Then if you modify `c[0]`, you also modify `__unnamed`, which is UB.

   This is documented at 6.4.5 "String literals":

   > 5 In translation phase 7, a byte or code of value zero is appended to each multibyte character sequence that results from a string literal or literals. The multibyte character sequence is then used to initialize an array of static storage duration and length just sufficient to contain the sequence. For character string literals, the array elements have type char, and are initialized with the individual bytes of the multibyte character sequence [...]
   >
   > 6 It is unspecified whether these arrays are distinct provided their elements have the appropriate values. If the program attempts to modify such an array, the behavior is undefined.

6.7.8/32 "Initialization" gives a direct example:

> EXAMPLE 8: The declaration
>
> ```
> char s[] = "abc", t[3] = "abc";
> ```
>
> defines "plain" char array objects `s` and `t` whose elements are initialized with character string literals.
>
> This declaration is identical to
>
> ```
> char s[] = { 'a', 'b', 'c', '\0' },
> t[] = { 'a', 'b', 'c' };
> ```
>
> The contents of the arrays are modifiable. On the other hand, the declaration
>
> ```
> char *p = "abc";
> ```
>
> defines `p` with type "pointer to char" and initializes it to point to an object with type "array of char" with length 4 whose elements are initialized with a character string literal. If an attempt is made to use `p` to modify the contents of the array, the behavior is undefined.

GCC 4.8 x86-64 ELF implementation

Program:

```
#include <stdio.h>

int main(void) {
    char *s = "abc";
    printf("%s\n", s);
    return 0;
}
```

Compile and decompile:

```
gcc -ggdb -std=c99 -c main.c
objdump -Sr main.o
```

Output contains:

```
 char *s = "abc";
8:  48 c7 45 f8 00 00 00    movq   $0x0,-0x8(%rbp)
f:  00
        c: R_X86_64_32S .rodata
```

Conclusion: GCC stores `char*` it in `.rodata` section, not in `.text`.

If we do the same for `char[]`:

```
char s[] = "abc";
```

we obtain:

```
17:   c7 45 f0 61 62 63 00    movl   $0x636261,-0x10(%rbp)
```

so it gets stored in the stack (relative to `%rbp`).

Note however that the default linker script puts `.rodata` and `.text` in the same segment, which has execute but no write permission. This can be observed with:

```
readelf -l a.out
```

which contains:

```
 Section to Segment mapping:
  Segment Sections...
   02     .text .rodata
```

Share Edit Follow Flag

10

You're not allowed to change the contents of a string constant, which is what the first `p` points to. The second `p` is an array initialized with a string constant, and you *can* change its contents.

Share Edit Follow Flag

8

For cases like this, the effect is the same: You end up passing the address of the first character in a string of characters.

The declarations are obviously not the same though.

The following sets aside memory for a string and also a character pointer, and then initializes the pointer to point to the first character in the string.

```
char *p = "hello";
```

While the following sets aside memory just for the string. So it can actually use less memory.

```
char p[10] = "hello";
```

Share Edit Follow Flag

5

From APUE, Section 5.14 :

```
char    good_template[] = "/tmp/dirXXXXXX"; /* right way */
char    *bad_template = "/tmp/dirXXXXXX"; /* wrong way*/
```

> ... For the first template, the name is allocated on the stack, because we use an array variable. For the second name, however, we use a pointer. In this case, only the memory for the pointer itself resides on the stack; the compiler arranges for the string to be stored in the read-only segment of the executable. When the `mkstemp` function tries to modify the string, a segmentation fault occurs.

The quoted text matches @Ciro Santilli 's explanation.

Share Edit Follow Flag

3

As far as I can remember, an array is actually a group of pointers. For example

```
p[1]== *(&p+1)
```

is a true statement

Share Edit Follow Flag

2

`char p[3] = "hello"` ? should be `char p[6] = "hello"` remember there is a '\0' char in the end of a "string" in C.

anyway, array in C is just a pointer to the first object of an adjust objects in the memory. the only different s are in semantics. while you can change the value of a pointer to point to a different location in the memory an array, after created, will always point to the same location.
also when using array the "new" and "delete" is automatically done for you.

Share Edit Follow Flag