Why do we not need an array size as a parameter while passing an array of characters to a function in C?

Joe Zbiciak

I have been programming since grade schoolAuthor
has **5.2K** answers and **36.8M** answer viewsUpdated 2y

Sometimes you do!

For example:

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

int main() {
  char arr1[5] = { 'H', 'e', 'l', 'l', 'o' };
  char arr2[6];

  // Note: sizeof(arr1) == 5, because sizeof(char) == 1
always.
  memcpy(arr2, arr1, sizeof(arr1));
        arr2[5] = '\0';

        puts(arr2);  // Prints 'Hello'
      }
```

That code won’t work without passing in the array size for `arr1`.
Now, if the array of `char` *happens* to hold a valid C string, *and* you’re using a string manipulation routine or other function that expects a C string, it will find the end of the string on its own. How? Valid C strings are terminated by a null character (`‘\0’`).
In my example above, `arr1` is an array of `char`, but it’s not a valid C string. I’m manipulating it with something that *isn’t* a string manipulation function. So, I have to pass in its length.
Not all arrays of `char` are C strings. Not all strings are C strings, either.
In particular, if you ever encounter utmp/wtmp records on a UNIX-style system, beware of the `char` arrays in its structure. They look like C strings, but they aren’t. Here’s an example version of `struct utmp` from the link above.

```
struct utmp {
    short   ut_type;              /* Type of record */
    pid_t   ut_pid;               /* PID of login process */
    char    ut_line[UT_LINESIZE]; /* Device name of tty - "/dev/" */
    char    ut_id[4];             /* Terminal name suffix,
                                     or inittab(5) ID */
    char    ut_user[UT_NAMESIZE]; /* Username */
    char    ut_host[UT_HOSTSIZE]; /* Hostname for remote login, or
                                     kernel version for run-level
                                           messages */
          struct  exit_status ut_exit;  /* Exit status of a process
                                           marked as DEAD_PROCESS; not
                                           used by Linux init(8) */
          /* The ut_session and ut_tv fields must be the same size when
             compiled 32- and 64-bit.  This allows data files and shared
             memory to be shared between 32- and 64-bit applications. */
      #if __WORDSIZE == 64 && defined __WORDSIZE_COMPAT32
          int32_t ut_session;           /* Session ID (getsid(2)),
                                           used for windowing */
          struct {
              int32_t tv_sec;           /* Seconds */
              int32_t tv_usec;          /* Microseconds */
          } ut_tv;                      /* Time entry was made */
      #else
          long   ut_session;            /* Session ID */
          struct timeval ut_tv;         /* Time entry was made */
      #endif

          int32_t ut_addr_v6[4];        /* Internet address of remote
                                           host; IPv4 address uses
                                           just ut_addr_v6[0] */
          char __unused[20];            /* Reserved for future use */
      };
```

See those fields named `ut_line`, `ut_user`, and `ut_host`?
They hold strings, but they aren't C strings. The commentary below the structure says the following:

String fields are terminated by a null byte ('\0') **if they are shorter than the size of the field.**

I bolded the important part. Those fields hold strings that aren't necessarily null-terminated. Those fields will behave like C strings right up until a string comes along that fills one of the fields entirely. At that point, things will break if your code assumes that field holds C string.

You really have to pay attention to what your arrays hold.

FWIW, this is the sort of field `strncpy`[1] was meant to fill. I wouldn't use `strncpy` for much of anything else, really. `strncpy` has the dubious distinction of being a string copy whose output is not guaranteed to be a valid C string, since it doesn't guarantee it'll null terminate the string. But, it'll pad out the field with 0s *if* the string's shorter than the output buffer.

Footnotes
[1]
strncpy, strncpy_s

Tom Crosley

M.S. in Computer Science, Illinois Institute of Technology Chicago - Illinois TechAuthor has **2.6K** answers and **23.7M** answer views2y

Because the C compiler doesn't care. Unlike other languages, C doesn't do any bounds checking, so it is the programmer's responsibility not to run past the end of an array. This is one of the ways programming in C can be dangerous.

In C, a reference to an array is treated exactly the same as a pointer. Thus, if we have

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

then `a[5]` and `*(a+5)` are exactly the same thing. Likewise, the address a+5 is the same as &a[5]. After initially allocating memory for the array `a`, the compiler doesn't care about the size anymore. So the function parameter can be declared as just `int a[]` or `int *a.`
Note that I haven't said whether this array of characters is a string or not. It could just be an array of ten signed bytes.

C doesn’t have an inherent string type. By convention (meaning all C library routines like strcpy, printf, etc. use this convention), strings are null terminated. So the length of the string is not explicitly specified in the array. However it could be — there is nothing that says the first byte of the array can’t be the string length (limiting the strings to a maximum length of 255 characters), and rewriting all of the library routines to use that convention.

If the array isn’t just one byte chars, the size of each element must be considered by the compiler. But the equivalency of an array index and a pointer remains the same. Say we have:

```
int b [10];
```

then once again `b[5]` and `*(b+5)` are exactly the same. Because of the way pointer arithmetic is handled, if an `int` is four bytes long, then `*(b+5)` points to an element 20 bytes past the beginning of the array, not 5.
If an array Is two-dimensional, then the compiler does need to know the size of one of the dimensions when passed a parameter. This is because a two-dimensional array is treated as an array of one dimensional arrays, not a pointer to a pointer (double pointer) which is different.

So if one has an 2-D array:

```
char c [4][3];
```

then the function parameter must be declared as either `int c[][3]` or `int (*c)[3]`.

Bailey Stoner

Studied at School of Hard Knocks (Graduated 1986)Author has **144** answers and **80.7K** answer views2y

In C, arrays can’t be inspected to get their lengths. So, either have to include a termination character in the data or you have to carry around a length variable to pass around so that everyone knows how big the array is.

If you don’t do this, then your application won’t know when to stop iterating the array and it will try to access memory that doesn’t exist. Take this for example:

```
int sum(int length, int* values) {
  int total = 0;
  unsigned int i;
  for (i = 0; i < length; ++i)
    total += values[i];
  return total;
}
```

If we didn’t have the `length` in this example, then we’d have absolutely no way of know when to stop looping through or array. The same thing goes for any data type and this includes char* because it is really just an array of char.

However, there is one case where this isn’t true. Some functions expect NULL-terminated strings. If you can guarantee that strings are NULL-terminated, then you can omit the length because you can search until the value is equal to ‘\0′ (aka NULL). When the value is NULL, you can break out of the function safely. However, it is important that you never send data into functions like this unless you know for certain that it is NULL-terminated or else you will accidentally loop beyond the length of the string and try to access memory that you may not be allowed to access, and you will get undefined behavior.

Anders Kaseorg

MIT PhD student in CS; Cofounder of Ksplice, Inc.Author
has **549** answers and **9.2M** answer viewsUpdated 7y

Related
Why doesn’t "sizeof(a)/sizeof(a[0])" work for an array passed as a parameter?

Originally Answered: Why doesn't sizeof(a)/sizeof(a[0]) doesn't work when applied for an array passed as parameters?

The behavior you found is actually a big wart in the C language. Whenever you declare a function that takes an array parameter, the C standard requires the compiler to ignore you and **change the parameter to a pointer**. So these declarations all behave like the first one:

- `void func(int *a)`
- `void func(int a[])`
- `void func(int a[5])`
- `typedef int array_plz[5];`
  `void func(array_plz a)`

`a` will be a pointer to `int` in all four cases. If you pass an array to `func`, it will immediately decay into a pointer to its first element. (On a 64-bit system, a 64-bit pointer is twice as large as a 32-bit `int`, so your `sizeof` ratio returns 2.)
This only happens in the context of function parameters. In general, arrays are not the same thing as pointers. In contexts other than function parameters, such as variables, `struct` members, array elements, or pointer targets, `int *`, `int[]`, `int[5]`, and `int[7]` are all different types with distinguishable behaviors; for instance, they have different sizes. (“The first step to learning C is understanding that pointers and arrays are the same thing. The second step is understanding that pointers and arrays are different.”)
The only purpose of this rule is to maintain backwards compatibility with historical compilers that did not support passing aggregate values as function arguments.

This does not mean that it’s impossible to pass an array to a function. You can get around this wart by embedding the array into a `struct` (this is basically the purpose of C++11’s `std::array`):

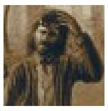
```
struct array_rly {
    int a[5];
};
void func(struct array_rly a)
{
    printf("%zd\n", sizeof(a.a)/sizeof(a.a[0]));  /* prints 5 */
}
```

or by passing a pointer to the array:

```
void func(const int (*a)[5])
{
    printf("%zd\n", sizeof(*a)/sizeof((*a)[0]));  /* prints 5 */
}
```

In case the array size isn't a compile-time constant, you can use the pointer-to-array technique with C99 variable-length arrays:

```
void func(int n, const int (*a)[n])
{
    printf("%zd\n", sizeof(*a)/sizeof((*a)[0]));  /* prints n */
}
```



Chris Reid

B.S in Computer Science & Electronics Technology (AAS), University of Alaska Anchorage (Graduated 2010)Author has **4.3K** answers and **7.5M** answer viewsUpdated 3y

Related
How can I find the number of elements in an array in C programming?

Save the size in a variable.

C arrays are inferior objects. They have no markers for size, and no operations that can be performed with them. They are only a contiguous range of bytes.
They are of fixed size at declaration, and any tricks used to change the size on the fly are dangerous moves with complications, and generally a poor design.

You must save the item count the array was declared with, or you risk exceeding the bounds of the array, which can cause all kinds of problems, like crashing the program, destroying the data, etc.

Always send in the upper bound of the array as a separate parameter into a function along with the array pointer.

I know that C strings violate this principle all over hell, but hey they are a legacy from the 1960s. C strings have as much chance of blowing up the system as any other array, we just learn to live with them.
`strlen()` is a terribly inefficient function (it has to iterate the whole string until it finds the first `'\0'` char, and that tells you nothing about the size of the memory allocation), but it's all we've got, so we go with it. `strlen` still segmentation faults on a `null` char pointer, all these years later, by the way.
If you find yourself trying to figure out how long an array is, you have already messed up. Rewrite the code before to fix the hole in the data, namely the bounds of the array. The array was declared with a size, so you can store it and retrieve it, so do that.

UNRELIABLE

The old trick of

```
size_t arraysize = sizeof(array)/sizeof(array[0]);
```

is an unreliable metric.

It is also quite dangerous.

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


int main( int argc, char** argv)
{


  char a[10] = {0};

        char* p = a + 5;

        size_t n = sizeof(a) / sizeof(a[0]);

        size_t n2 = sizeof(p) / sizeof(p[0]);

        fprintf(stdout,"a:\t%zu\n", n);

        fprintf(stdout,"p:\t%zu\n", n2);



        return 0;
      }

      a: 10
      p: 8
```

So n2 is wrong. because p is a pointer in an array, but can be used as an array
and where does the value 8 come from? 8 what? Are we counting byte ratios or bytes or elements? Bleeechhhh! *Just don't.*

Structured arrays

```
typedef struct _
{
   double dval[100];
}Darray100;
```

This is an interesting beast. It is elevated to a complete type, yet is just an array of doubles.
But the compiler knows its size, and how to make one. You can copy it. You can assign it. You can decompose it into its elements. You can force it into being a plain array and index it.

```
Darray100 d;  /* allocates 100 doubles */

#define TOARRAY(X)  (double*)(X)

size_t i=0;
double* dp = TOARRAY(d);

for (i = 0; i < 100; i++) dp[i]= 0.00; /* initialize */

      Darray100 x = d;  /* binary copy entire struct */
```

I have never really seen this in real code.

Dynamic Allocation

Of course the most dangerous thing is to allocate an array with malloc or calloc and not keep the size and the original pointer around. If you lose the pointer, you are screwed. If you don't know the size you are screwed. The tiny C runtime keeps a table of the allocations and sizes, so free will work pretty reliably if you maintain the original pointer used for the malloc call.
It all hinges on knowing the byte-size of your object, and how many you are allocating.

```
/* 80 char */

char* buffer = (char*) malloc( 80 *sizeof(char));
memset(buffer, '\0', 80); /* memset works with bytes */

/* calloc is designed to be a bit 'cleaner' than malloc
```

```
    by zeroing out the memory, and doing the multiplication */

size_t intsz = sizeof(int);
     size_t numelements = 1000;
     int* x = (int*) calloc( numelements, intsz); /* 4000 bytes, 1000 ints */
```