Array of Strings | Coursera

coursera.org/learn/pointers-arrays-recursion/supplement/DYMu6/array-of-strings

Array of Strings

Array of Strings

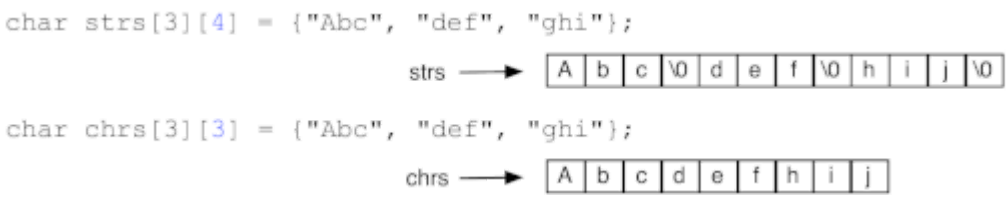
As we mentioned earlier, an array of strings is inherently a multidimensional array of characters, as strings themselves are really just arrays of characters. Accordingly, all the same rules apply to arrays of strings, and we can use either representation that we want. However, as arrays of strings are fairly important (among other things, the program can access its command line arguments via an array of strings), it is worth discussing them explicitly.

Consider the following two statements, each of which declares a multidimensional array of **char**s, and initializes it with a braced array of string literals:

```
2

char chrs[3][3] = {"Abc", "def", "ghi"};
```

Observe that the difference between the two declarations is the size of the second dimension of the array—which is 4 in the first statement, and 3 in the second. The first statement (which declares *strs*) includes space for the null terminator, which is required to make the sequence of characters a valid string. The second statement, which declares *chrs*, does not include such space and only stores the characters that were written (with no null terminator). The figure below illustrates the effects of the two statements.



This second statement is correct if (and only if) we intend to use *chrs* only as a multidimensional array of characters, and not use its elements for anything which expects a null terminated string. As the second makes a valid multidimensional array of *chars*, it is not illegal, and will not produce an error or a warning. This behavior is much the same as we discussed previously, for just declaring arrays of characters and initializing them

from string literals. However, a significant difference is that in the multidimensional case, we cannot omit the size from the second dimension (which is the number of characters in each string), as C allows us to omit only the first dimensions of a multidimensional array.

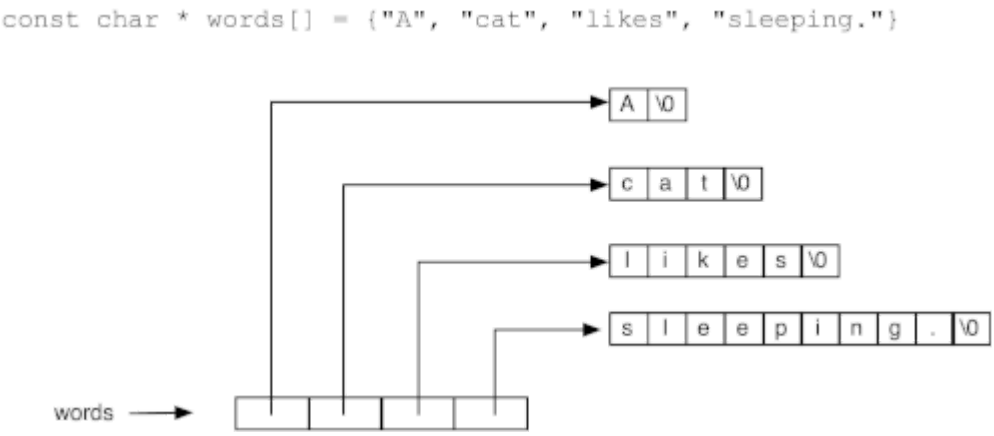
If you declare a multidimensional array of *char*s to hold strings of different lengths, then you must size the second dimension according to the length of the longest string. For example, we might declare the following array:

```
1

2

char words[4][10] = {"A", "cat", "likes", "sleeping."};
```

In this example, words requires 40 characters of storage despite the fact that the strings used to initialize it only occupy 22 characters. This representation wastes some space. While that waste may not be significant in this example, if we were instead looking at millions of strings with lengths that vary greatly, we might be wasting megabytes.



This figure illustrates an array of strings (i.e., an array of pointers to sequences of **char**s ending in a null terminator).

We might instead use the array-of-pointers representation for an array of strings. As we previously discussed, representing multidimensional data with an array of pointers allows us to have items of different lengths, which naturally solves the problem of wasted space. To represent our array of strings in this fashion, we might declare and initialize words as follows:

```
1
```

```
2
```

```
const char * words[] = {"A", "cat", "likes", "sleeping."};
```

Observe that here, we declare words as an array of **const char** *s—the elements of the array are pointers to **const char**s, and thus the **char**s they point to may not be modified. We should include the **const** (and *must* include it to be **const**-correct) as we have indicated that words should be initialized to pointers to string literals, which are in read-only memory. The figure above illustrates the layout of *words*.

We will note that it is common to end an array of strings with a NULL pointer, such as this:

```
1
```

```
const char * words2[] = {"A", "cat", "likes", "sleeping.", NULL};
```

This convention is common, as it allows for one to write loops which iterate over the array without knowing *a priori* how many elements are in the array. Instead, the loop can have a condition which checks for NULL, such as this:

```
3
```

```
4
```

```
5
```

```
6
```

```
1
```

```
2
```

```
  printf("%s ", *ptr);
```

```
  ptr++;
}
printf("\n");
const char ** ptr = words2;
while (*ptr != NULL) {
```

It would be a beneficial exercise to execute this code by hand to make sure you understand all of the concepts involved. You can then put it into a source file (with appropriate *includes*, and inside of *main*) to make sure you derived the correct answer.