C: pointer to array of pointers to structures (allocation/deallocation issues)

Asked 11 years, 9 months ago Modified 10 months ago Viewed 146k times

41

I've been getting back into C for something, but I'm having trouble remembering much of how this memory management works. I'd like to have a pointer to an array of pointers to structures.

Say I have:

```
struct Test {
   int data;
};
```

Then the array:

```
struct Test **array1;
```

Is this correct? My issue is working with this thing. So each pointer in the array points to something that is allocated separately. But I think I need to do this first:

```
array1 = malloc(MAX * sizeof(struct Test *));
```

I am having trouble understanding the above. Do I need to do this, and why do I need to do this? In particular, what does it mean to allocate memory for pointers if I am going to be allocating memory for each thing that the pointer points to?

Say now I have pointer to an array of pointers to structures. I now want it to point to the same array that I've created earlier.

```
struct Test **array2;
```

Do I need to allocate room for pointers like I did above, or can I just do:

```
array2 = array1
```

c pointers memory memory-management

Share

Improve this question

Follow

edited Mar 14, 2013 at 0:17

Michael Anderson
73.3k ● 8 ● 146 ● 192

asked Mar 13, 2013 at 22:27

DillPixel
965 ● 1 ● 14 ● 21

take a look at stackoverflow.com/questions/11421884/… – Richard Chambers Mar 13, 2013 at 22:31

Do you want an actual array of pointers to the structs? As in a declared array where you allocate each element with a struct? – teppic Mar 13, 2013 at 22:35

Well I want a pointer to an array where I can perform what you said. – DillPixel Mar 13, 2013 at 22:36

I know. But do you want a *real* array to hold those? Rather than just a pointer to a block of memory I mean. – teppic Mar 13, 2013 at 22:37

1 It's just simpler with a proper array - I can post an example if you want. – teppic Mar 13, 2013 at 22:39

4 Answers

Sorted by: Highest score (default)

110

Allocated Array

With an allocated array it's straightforward enough to follow.

Declare your array of pointers. Each element in this array points to a `struct Test`:

```
struct Test *array[50];
```

Then allocate and assign the pointers to the structures however you want. Using a loop would be simple:

```
array[n] = malloc(sizeof(struct Test));
```

Then declare a pointer to this array:

```
                               // an explicit pointer to an array
struct Test *(*p)[] = &array;  // of pointers to structs
```

This allows you to use `(*p)[n]->data`; to reference the nth member.

Don't worry if this stuff is confusing. It's probably the most difficult aspect of C.

Dynamic Linear Array

If you just want to allocate a block of structs (effectively an array of structs, *not* pointers to structs), and have a pointer to the block, you can do it more easily:

```
struct Test *p = malloc(100 * sizeof(struct Test));  // allocates 100 linear
                                                     // structs
```

You can then point to this pointer:

```
struct Test **pp = &p
```

You don't have an array of pointers to structs any more, but it simplifies the whole thing considerably.

Dynamic Array of Dynamically Allocated Structs

The most flexible, but not often needed. It's very similar to the first example, but requires an extra allocation. I've written a complete program to demonstrate this that should compile fine.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct Test {
    int data;
};

int main(int argc, char **argv)
{
    srand(time(NULL));

    // allocate 100 pointers, effectively an array
    struct Test **t_array = malloc(100 * sizeof(struct Test *));

    // allocate 100 structs and have the array point to them
    for (int i = 0; i < 100; i++) {
        t_array[i] = malloc(sizeof(struct Test));
    }

    // lets fill each Test.data with a random number!
    for (int i = 0; i < 100; i++) {
        t_array[i]->data = rand() % 100;
    }

    // now define a pointer to the array
    struct Test ***p = &t_array;
    printf("p points to an array of pointers.\n"
       "The third element of the array points to a structure,\n"
       "and the data member of that structure is: %d\n", (*p)[2]->data);
    return 0;
}
```

Output:

```
> p points to an array of pointers.
> The third element of the array points to a structure,
> and the data member of that structure is: 49
```

Or the whole set:

```
for (int i = 0; i < 100; i++) {
    if (i % 10 == 0)
        printf("\n");
    printf("%3d ", (*p)[i]->data);
}

 35  66  40  24  32  27  39  64  65  26
 32  30  72  84  85  95  14  25  11  40
 30  16  47  21  80  57  25  34  47  19
 56  82  38  96   6  22  76  97  87  93
 75  19  24  47  55   9  43  69  86   6
 61  17  23   8  38  55  65  16  90  12
 87  46  46  25  42   4  48  70  53  35
 64  29   6  40  76  13   1  71  82  88
 78  44  57  53   4  47   8  70  63  98
 34  51  44  33  28  39  37  76   9  91
```

Dynamic Pointer Array of Single-Dynamic Allocated Structs

This last example is rather specific. It is a dynamic array of pointers as we've seen in previous examples, but unlike those, the elements are all allocated in a *single* allocation. This has its uses, most notable for sorting data in different configurations while leaving the original allocation undisturbed.

We start by allocating a single block of elements as we do in the most basic single-block allocation:

```
struct Test *arr = malloc(N*sizeof(*arr));
```

Now we allocate a *separate* block of pointers:

```
struct Test **ptrs = malloc(N*sizeof(*ptrs));
```

We then populate each slot in our pointer list with the address of one of our original array. Since pointer arithmetic allows us to move from element to element address, this is straight-forward:

```
for (int i=0;i<N;++i)
    ptrs[i] = arr+i;
```

At this point the following both refer to the same element field

```
arr[1].data = 1;
ptrs[1]->data = 1;
```

And after review the above, I hope it is clear *why*.

When we're done with the pointer array and the original block array they are freed as:

```
free(ptrs);
free(arr);
```

Note: we do NOT free each item in the `ptrs[]` array individually. That is not how they were allocated. They were allocated as a single block (pointed to by `arr`), and that is how they should be freed.

So why would someone want to do this? Several reasons.

First, it radically reduces the number of memory allocation calls. Rather then `N+1` (one for the pointer array, N for individual structures) you now have only **two**: one for the array block, and one for the pointer array. Memory allocations are one of the most expensive operations a program can request, and where possible, it is desirable to minimize them (note: file IO is another, fyi).

Another reason: Multiple representations of the same base array of data. Suppose you wanted to sort the data both ascending and descending, and have both sorted representations available *at the same time*. You could duplicate the data array, but that would require a lot of copying and eat significant memory usage. Instead, just allocate an extra pointer array and fill it with addresses from the base array, then sort that pointer array. This has especially significant benefits when the data being sorted is large (perhaps kilobytes, or even larger, per item) The original items remain in their original locations in the base array, but now you have a very efficient mechanism in which you can sort them without having to actually *move* them. You sort the array of pointers to items; the items don't get moved at all.

I realize this is an awful lot to take in, but pointer usage is critical to understanding the many powerful things you can do with the C language, so hit the books and keep refreshing your memory. It will come back.

Share edited Mar 14, 2013 at 2:21 answered Mar 13, 2013 at 22:45

Improve this answer

Follow

WhozCraig
66.1k ● 11 ● 79 ● 147

teppic
8,175 ● 2 ● 25 ● 38

Say I have another structure Test2, that holds this pointer to array. How would I allocate that on the heap? struct Test2 { struct Test *array[50]; }; struct Test2 *container = malloc(sizeof(Test2)) Is that enough? – DillPixel Mar 13, 2013 at 22:57

@DillPixel: That's declaring the array of pointers itself in the 2nd struct. If you just want a struct to point to the array,you only need to define a pointer. (This is starting to make my head hurt) – teppic Mar 13, 2013 at 23:04

2 Is there terminology for each type of dynamic allocation mentioned here? I would like to be able to google search related things. Prior to this I've somehow understood "Dynamic Linear Array" and "Dynamic Array of Dynamically Allocated Structs" but don't know how to express them in Google search term other than Dynamic Array Allocation. – raymai97 Jun 3, 2017 at 12:47

5

It may be better to declare an actual array, as others have suggested, but your question seems to be more about memory management so I'll discuss that.

```
struct Test **array1;
```

This is a pointer to the address of a `struct Test`. (Not a pointer to the struct itself; it's a pointer to a memory location that holds the *address* of the struct.) The declaration allocates memory for the pointer, but not for the items it points to. Since an array can be accessed via pointers, you can work with `*array1` as a pointer to an array whose elements are of type `struct Test`. But there is not yet an actual array for it to point to.

```
array1 = malloc(MAX * sizeof(struct Test *));
```

This allocates memory to hold `MAX` pointers to items of type `struct Test`. Again, it does *not* allocate memory for the structs themselves; only for a list of pointers. But now you can treat `array` as a pointer to an allocated array of pointers.

In order to use `array1`, you need to create the actual structs. You can do this by simply declaring each struct with

```
struct Test testStruct0;  // Declare a struct.
struct Test testStruct1;
array1[0] = &testStruct0;  // Point to the struct.
array1[1] = &testStruct1;
```

You can also allocate the structs on the heap:

```
for (int i=0; i<MAX; ++i) {
  array1[i] = malloc(sizeof(struct Test));
}
```

Once you've allocated memory, you can create a new variable that points to the same list of structs:

```
struct Test **array2 = array1;
```

You don't need to allocate any additional memory, because `array2` points to the same memory you've allocated to `array1`.

Sometimes you *want* to have a pointer to a list of pointers, but unless you're doing something fancy, you may be able to use

```
struct Test *array1 = malloc(MAX * sizeof(struct Test));  // Pointer to MAX
structs
```

This declares the pointer `array1`, allocated enough memory for `MAX` structures, and points `array1` to that memory. Now you can access the structs like this:

```
struct Test testStruct0 = array1[0];     // Copies the 0th struct.
struct Test testStruct0a= *array1;       // Copies the 0th struct, as above.
struct Test *ptrStruct0 = array1;        // Points to the 0th struct.

struct Test testStruct1 = array1[1];     // Copies the 1st struct.
struct Test testStruct1a= *(array1 + 1); // Copies the 1st struct, as above.
struct Test *ptrStruct1 = array1 + 1;    // Points to the 1st struct.
struct Test *ptrStruct1 = &array1[1];    // Points to the 1st struct, as above.
```

So what's the difference? A few things. Clearly, the first method requires you to allocate memory for the pointers, and then allocate additional space for the structs themselves; the second lets you get away with one `malloc()` call. What does the extra work buy you?

Since the first method gives you an actual array of pointers to `Test` structs, each pointer can point to any `Test` struct, anywhere in memory; they needn't be contiguous. Moreover, you can allocate and free the memory for each actual `Test` struct as necessary, and you can reassign the pointers. So, for example, you can swap two structures by simply exchanging their pointers:

```
struct Test *tmp = array1[2];  // Save the pointer to one struct.
array1[2] = array1[5];         // Aim the pointer at a different struct.
```

```
array1[5] = tmp;              // Aim the other pointer at the original struct.
```

On the other hand, the second method allocates a single contiguous block of memory for all of the `Test` structs and partitions it into `MAX` items. And each element in the array resides at a fixed position; the only way to swap two structures is to copy them.

Pointers are one of the most useful constructs in C, but they can also be among the most difficult to understand. If you plan to continue using C, it'll probably be a worthwhile investment to spend some time playing with pointers, arrays, and a debugger until you're comfortable with them.

Good luck!

Share Improve this answer Follow

answered Mar 13, 2013 at 23:18

Adam Liss
48.3k ● 13 ● 113 ● 152

5

I suggest that you build this out a layer at a time using typdefs to create layers of types. By doing so, the different types needed will be much clearer.

For instance:

```
typedef struct Test {
   int data;
} TestType;

typedef  TestType * PTestType;
```

This will create two new types, one for the struct and one for a pointer to the struct.

So next if you want an array of the structs then you would use:

```
TestType array[20];  // creates an array of 20 of the structs
```

If you want an array of pointers to the structs then you would use:

```
PTestType array2[20];  // creates an array of 20 of pointers to the struct
```

Then if you want to allocate structs into the array you would do something like:

```
PTestType  array2[20];  // creates an array of 20 of pointers to the struct
// allocate memory for the structs and put their addresses into the array of
pointers.
for (int i = 0; i < 20; i++) {
        array2 [i] = malloc (sizeof(TestType));
    }
```

C does not allow you to assign one array to another. You have a couple of options. You can instead use a loop to assign each element of one array to an element of the other. Or you can use the memory functions `memmove()` and/or `memcpy()` move the contents of memory of one array to the memory area of another.

Or if it is a static array within a `struct`, you can assign one `struct` variable to another thereby performing an array assignment as one element of a `struct` assignment. However this last option won't work if the array is a pointer to a list since a `struct` assignment is a basic memory area to memory area copy similar to using the `memmov()` or `memcpy()` functions so the value of the pointer in the `struct` would be copied but both variables would contain a pointer whose value is the same address hence both pointing to the same area of memory which was not copied.

EDIT: Another Interesting Approach

Another approach would be a more object oriented approach in which you encapsulate a few things. For instance using the same layers of types we create two types:

```
typedef struct _TestData {
    struct {
        int myData;    // one or more data elements for each element of the
pBlob array
    } *pBlob;
    int nStructs;         // count of number of elements in the pBlob array
} TestData;

typedef TestData *PTestData;
```

Next we have a helper function which we use to create the object, named appropriately enough `CreateTestData (int nArrayCount)`.

```
PTestData  CreateTestData (int nCount)
{
    PTestData ret;

    // allocate the memory for the object. we allocate in a single piece of
memory
    // the management area as well as the array itself.  We get the sizeof ()
the
    // struct that is referenced through the pBlob member of TestData and
multiply
    // the size of the struct by the number of array elements we want to have.
    ret = malloc (sizeof(TestData) + sizeof(*(ret->pBlob)) * nCount);
    if (ret) {   // make sure the malloc () worked.
            // the actual array will begin after the end of the TestData struct
        ret->pBlob = (void *)(ret + 1);   // set the beginning of the array
        ret->nStructs = nCount;           // set the number of array elements
    }

    return ret;
}
```

Now we can use our new object as in the source code segment below. It should check that the pointer returned from CreateTestData() is valid however this is really just to show what could be done.

```
PTestData  go = CreateTestData (20);
{
    int i = 0;
    for (i = 0; i < go->nStructs; i++) {
        go->pBlob[i].myData = i;
    }
}
```

In a truly dynamic environment you may also want to have a `ReallocTestData(PTestData p)` function that would reallocate a `TestData` object in order to modify the size of the array contained in the object.

With this approach, when you are done with a particular TestData object, you can just free the object as in `free (go)` and the object and its array are both freed at the same time.

Edit: Extending Further

With this encapsulated type we can now do a few other interesting things. For instance, we can have a copy function, `PTestType CreateCopyTestData (PTestType pSrc)` which would create a new instance and then copy the argument to a new object. In the following example, we reuse the function `PTestType CreateTestData (int nCount)` that will create an instance of our type, using the size of the object we are copying. After doing the create of the new object, we make a copy of the data from the source object. The final step is to fix up the pointer which in the source object points to its data area so that pointer in the new object now points to the data area of itself rather than the data area of the old object.

```
PTestType CreateCopyTestData (PTestType pSrc)
{
    PTestType pReturn = 0;

    if (pSrc) {
        pReturn = CreateTestData (pSrc->nStructs);

        if (pReturn) {
            memcpy (pReturn, pSrc, sizeof(pTestType) + pSrc->nStructs *
sizeof(*(pSrc->pBlob)));
            pReturn->pBlob = (void *)(pReturn + 1);   // set the beginning of
the array
        }
    }

    return pReturn;
}
```

Share

Improve this answer

Follow

edited Feb 23 at 4:55

answered Mar 13, 2013 at 22:42

Richard Chambers
17.5k ● 4 ● 91 ● 118

2

Structs are not very different from other objects. Let's start with characters:

```
char *p;
p = malloc (CNT * sizeof *p);
```

*p is a character, so `sizeof *p` is sizeof (char) == 1; we allocated CNT characters. Next:

```
char **pp;
pp = malloc (CNT * sizeof *pp);
```

p is a pointer to character, so `sizeof *pp` is sizeof (char). We allocated CNT pointers. Next:

```
struct something *p;
p = malloc (CNT * sizeof *p);
```

*p is a struct something, so `sizeof *p` is sizeof (struct something). We allocated CNT struct somethings. Next:

```
struct something **pp;
pp = malloc (CNT * sizeof *pp);
```

pp is a pointer to struct, so `sizeof *pp` is sizeof (struct something). We allocated CNT pointers.

Share

Improve this answer

Follow

edited Oct 23, 2016 at 10:55

answered Mar 14, 2013 at 0:17

wildplasser
44.2k ● 9 ● 71 ● 116

@Yar It could be. It could also be 4, or maybe even 2... It is irrelevant. It is also the reason why `sizeof` exists. – wildplasser Oct 23, 2016 at 10:54