

Re: manipulating void* in array

Source: http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2005-07/msg01758.html

- *From:* "Netocrat" <netocrat@xxxxxxxxxxxx>
 - *Date:* 19 Jul 2005 01:28:49 -0700
-

On Mon, 18 Jul 2005 06:55:52 -0700, Stijn van Dongen wrote:

> Dave Thompson wrote:

>> On 13 Jul 2005 15:51:04 -0700, "Stijn van Dongen" <svd@xxxxxxxxxxxx>

>> wrote:

<snip>

>>> Now I would like to have a function, say `hash_keys`, which returns all
>>> keys in the hash as a pointer to a malloced array of `void*`. However,
>>> there is not really a type I can use.

The correct type is `void**` as Dave pointed out.

`struct foo **` would be preferable but I understand that your function needs to be generic.

>>> I see two possible solutions.

>>> The first is to typedef a struct containing a void pointer:

>>>

>>> typedef struct {

>>> void* pp;

>>> } genpp;

>>>

>>> and have `hash_keys` return `genpp*` (the size could be written in an

>>> `int*` argument

>>> to `hash_keys`). This would make accessing the keys cumbersome.

>>>

>>> Solution 2) is illegal C (I think), but it's tempting. `hash_keys`

>>> would construct an array of `void*`-sized elements and copy its key

>>> pointers there using `char*` arithmetic.

But if I follow correctly, the key pointers are already `void*` type, so you can use a direct assignment rather than byte-by-byte copying.

>>> It would return `void*` and the

>>> caller would cast it to `foo**`.

As Dave pointed out, this is not a portable cast.

Re: manipulating void* in array

>>> For one thing, this assumes
>>> sizeof(void*) is sizeof(foo*) for all possible foo.

You're right that that's not a portable assumption. But how does foo* come into it? I thought your key pointer type was void*.

>>> For another, I
>>> see nothing in the standard on void* that would support any of this.
>>> However, it feels as if approach 2) is conceptually extremely similar
>>> to the first and I suspect it would work on nearly all
>>> platforms/compilers.
>>>
>>> An array of void*, addressed as void**, and an array of struct
>>> containing only void*, addressed as genpp*, are indeed similar. And in
>>> fact are very likely (though not guaranteed) to be laid out the same.
>>>
>>> I think the array of void* has to be addressed as void*; AFAIK there is
>>> no type 'void**' in C (as you cannot dereference a void* pointer).

A void ** type is valid. The result of dereferencing a void ** pointer is a void * pointer, which is allowed.

Your second solution is workable.

To expand on Dave's suggestion, here is how to modify your approach:

a) return void**, not void*
b) copy the pointers directly rather than byte-by-byte. If a conversion is required (afaict it isn't as the src and dest are both void*) then rely on these properties of void* pointers (N869, 6.3.2.3):
"A pointer to void may be converted to or from a pointer to any incomplete or object type. A pointer to any incomplete or object type may be converted to a pointer to void and back again; the result shall compare equal to the original pointer."
c) don't cast the return to foo** (as Dave said this is not portable); instead cast an element at a time:

```
struct foo *my_foo_ptr = returned_void_ptr_array[i]; /* cast not req'd*/  
struct foo my_foo_object = *my_foo_ptr;
```

or

```
struct foo my_foo_object = *(struct foo *)returned_void_ptr_array[i];  
/* cast required */
```

>>> If so, the logical thing to do is to create and return an array of
>>> void*, and the caller, who presumably knows the type of what was (or
>>> should have been) stored, then converts that to a foo* before using it,
>>> either by explicitly casting or implicitly by assigning to its own
>>> (copy) pointer.

Re: manipulating void* in array

Re: manipulating void* in array

>
> The reason why I don't want to do that is because it implies I'll have a
> whole array of *copies* of my foo things. Given any one foo (presumably
> some struct), I never want to duplicate it, and only pass pointers to it
> around. Having copies around would cause endless trouble with ownership
> issues. In the current setup returning foo* is also impossible since the
> hash routines never (can) dereference their void* key arguments. Which
> is a good thing. It could be achieved by passing another callback to the
> hash table creation routine of course. But I rather have a foo** array
> or a genpp* array as described above.

The void ** approach doesn't require copies, but depending on usage it may require typecasts.

If you can't accept the typecasting, then yes, a callback will be necessary.

Something like (untested code):

- 1) declare your return array in hash_keys as
void *retarray = NULL; /* NOT void ** since this implies retarray[2] is OK */
- 2) define your callback to take void **cbretarray, void *new_element, int num_elements
- 3) pass &retarray as cbretarray
- 4) within the callback:

```
void *orgmem = *cbretarray; /* in case realloc fails */
*cbretarray = realloc(*cbretarray, (num_elements + 1) * sizeof(struct
foo *));
if (!(*cbretarray) )
; /* handle out of mem; original memory pointed to by orgmem */
else
((struct foo **)*cbretarray)[num_elements] = new_element;
/* applying cast to new_element is unnecessary */
```

Then declare void *hash_keys(..) and call it as
struct foo ** my_foo_array = hash_keys(..) /* no cast req'd */

You may choose to pass num_elements as a pointer and to increment its dereferenced value within the callback only if realloc succeeds.

.

-
- *Follow-Ups:*
 - ◆ *Re: manipulating void* in array*
◇ From: Stijn van Dongen
 - *References:*
 - ◆ *manipulating void* in array*

Re: manipulating void* in array

◇ *From:* Stijn van Dongen

◆ ***Re: manipulating void* in array***

◇ *From:* Dave Thompson

◆ ***Re: manipulating void* in array***

◇ *From:* Stijn van Dongen

- Prev by Date: ***Re: World Smallest Program***
- Next by Date: ***Re: World Smallest Program***
- Previous by thread: ***Re: manipulating void* in array***
- Next by thread: ***Re: manipulating void* in array***
- Index(es):
 - ◆ ***Date***
 - ◆ ***Thread***