

Re: How to get the memory block size which allocated by new operator?

## Re: How to get the memory block size which allocated by new operator?

---

*Source:* <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.language/2006-09/msg00282.html>

---

- *From:* "space6" <[ispacex@xxxxxxxxxx](mailto:ispacex@xxxxxxxxxx)>
  - *Date:* 10 Sep 2006 19:27:19 -0700
- 

Alan Carre wrote:

"space6" wrote:

In my project, it seemed there is memory leak somewhere. I can't find the place even by using Numega BoundCheck. then I find a idea. I think I can use a global variable of integer to save the memory that not released. When allocate memory by new operator, I will add the size to the global variable, and when delete memory, I will sub the size. And I always output the global variable by OutputDebugString() API.

How do you know the size of the memory block you are deleting without a complete map of the entire allocation set?

I successd at Debug mode. The code to get a buffer's size which allocated by new operator as following:

```
int GetNewBufferSize(void *p)
{

    if ( p )
#ifdef _Debug
        return *(int*)((char *)p-16);
```

This is not strictly true. It depends on how your compiler implements new and "new[]" which may be the same or different and the location of the memory block data may not be 16 chars to the left of 'p' (though, if you say so, it probably often is). Your code is not portable though.

Re: How to get the memory block size which allocated by new operator?

```
#else
//How to get the memory size in release mode ?
```

Again no gurantees. For all you know new could use a HASH table mapping of PVOID's to some complicated structure with all kinds of information. No, your best bet is the following (below):

```
#endif
else
return 0;
}
```

A GREAT step is to write a memory manager yourself. Instead of using malloc and free, or new and delete, funnel all allocations through your own functions (you can globally override the new/free operators for this reason). #undef malloc in your basic.h, toss the phrase out of existence. Or if you can't #undef it, #define it to point to your memory manager instead. With new and delete you can simply override the defaults.

Steve Maguire, who used to be Director of Development at Msoft wrote an excellent book (written for C, no C++) on this very topic. He wrote down a set of memory management routines that are far superiour to malloc, realloc, strdup, free and so on, and developed a detailed (though simple to understand) memory management library (the full source is in the book).

His memory management utilities automatically catch memory leaks, lost pointers, can constantly monitor the integrity of the system and many other things. For one thing (asside from catching memory leaks) it allows you the flexibility to simulate memory failures in order to test your code for handling them. Steve was a great programmer and writer and if you or anyone can get a copy of his book (Writing Solid Code) i strongly urge you to do so. It's well worth the purchase price. I can't help it... I have to give one small sample from the book, maybe it will spark some interest (2 small sections if you want to read them):

=====

#### JUST A LITTLE EXTRA THOUGHT

Programmers know when they're combining multiple outputs into a single return value, so acting on the suggestion [of not doing it] above is easy—they just stop doing it. In other cases, though, an interface can seem fine but, like the Trojan horse, contain hidden danger. Take a look at this code to change the size of a memory block:

```
pbBuf = (byte*) realloc (pbBuf , sizeNew);
i f (pbBuf != NULL)
use / initialize the larger buffer
```

## Re: How to get the memory block size which allocated by new operator?

Do you see what's wrong with this? If you don't, don't worry—the bug is serious, but it's subtle, and very few people spot it unless they're given a hint. So, here's a hint: If `pbBuf` is the only pointer to the block about to be resized, what happens if the call to `realloc` fails? The answer is that `NULL` is stuffed into `pbBuf` when `realloc` returns, destroying the only pointer to the original block [alan added: which remains allocated]. The code creates lost memory blocks.

Here's a question: How many times do you want to resize a block and store the pointer to the new block in a different variable? I'd imagine about as often as you'd want to drive to a restaurant in one car and leave in another. Sure, there are cases in which you want to store the new pointer in a different variable, but normally, if you change the size of a block, you want to update the original pointer. That's why programmers so often fall into `realloc`'s trap. `realloc` has a candy-machine [described above] interface. Ideally, `realloc` would always return an error code and a pointer to the memory block regardless of whether the block was expanded. That's two separate outputs. Let's take another look at `fResizeMemory`, the cover function for `realloc` that I talked about in Chapter 3. Here it is again, stripped of all the debug code:

```
flag fResizeMemory(void **ppv, size_t sizeNew)
{
    byte **ppb = ( byte **)ppv;
    byte *pbNew;
    pbNew = ( byte* ) realloc ( *ppb , sizeNew);
    if (pbNew != NULL)
        *ppb = pbNew;
    return (pbNew != NULL);
}
```

Take a look at the `if` statement in the code above it ensures that the original pointer is never destroyed. If you rewrote the `realloc` code at the start of this section using `fResizeMemory`, you would have

```
if (fResizeMemory(&pbBuf, sizeNew))
    use/initialize the larger buffer
```

In this case, if the attempt to resize the block fails, `pbBuf` is left untouched and continues to point to the original block; `pbBuf` is not set to `NULL`. That's exactly the behavior you want. So here's a question: "How likely is it that a programmer will lose a memory block using `fResizeMemory`?" Here's another: "How likely is it that a programmer will forget to handle `fResizeMemory`'s error condition?"

Another interesting point to note is that programmers who habitually follow the earlier suggestion in this chapter—"Don't bury error codes in return values"—would never design an interface such as `realloc`'s. Their first attempt would be more like `fResizeMemory`'s—and so wouldn't have `realloc`'s

Re: How to get the memory block size which allocated by new operator?

"lost block" problem. The recommendations in this book build on each other and interact in ways that you might not expect. This is an example of that happening.

But separating your outputs is not always going to save you from designing interfaces with hidden traps. I wish I could offer a better piece of advice, but the only sure way to catch such hidden traps is to stop and think about your design. The best approach is to examine every possible combination of inputs and outputs and look for side effects that can cause problems. I know this can sometimes be tedious, but remember, it's relatively cheap for you to take the extra time up front to think about this. The worst thing you can do is to skip this step and force who knows how many other programmers into tracking down and fixing bugs caused by a poorly designed interface.

Imagine how much total time has been wasted by programmers all over the world who have been forced to track down bugs caused by the interface traps of `getchar`, `malloc`, and `realloc`—to say nothing of all the functions that have been written using one of these three as a model. It's a sobering amount of time.

#### THE ONE FUNCTION MEMORY MANAGER

Although I spent a lot of time talking about the `realloc` function in Chapter 3, I didn't cover many of its more bizarre aspects. If you pull out your C library manual and look up the full description of `realloc`, you'll find something like this:

```
void *realloc(void *pv, size_t size);
```

`realloc` changes the size of a previously allocated memory block. The contents of the block are preserved up to the lesser of the new and old block sizes.

If the new size of the block is smaller than the old size, `realloc` releases the unwanted memory at the tail end of the block and `pv` is returned unchanged.

If the new size is larger than the old size, the expanded block may be allocated at a new address and the contents of the original block copied to the new location. A pointer to the expanded block is returned, and the extended part of the block is left uninitialized.

If you attempt to expand a block and `realloc` cannot satisfy the request, `NULL` is returned. `realloc` will always succeed when you shrink a block.

If `pv` is `NULL`, then `realloc` behaves as though you called `malloc(size)` and returns a pointer to a newly allocated block, or `NULL` if the request cannot be satisfied.

Re: How to get the memory block size which allocated by new operator?

If the new size is 0 and pv is not NULL, then realloc behaves as though you called free(pv) and NULL is always returned. If pv is NULL and size is 0, the result is undefined.

Whew! realloc is a prime example of implementation overkill—it's a complete memory manager in just one function. Why do you need malloc? Why do you need free? realloc does it all.

There are several good reasons why you should not design functions this way. First, you can't expect programmers to use such a function safely. There are so many details that even experienced programmers don't know them all. If you doubt this, take a survey and tally how many programmers know that passing a NULL pointer to realloc simulates a call to malloc. Tally how many know that passing a 0 size is the same as calling free. True, this is fairly arcane behavior, so ask them a question they must know the answer to if they hope to avoid bugs. Ask them what happens when they call realloc to expand a block. Do they know that the block can move?

Here's another problem with realloc: You know it's possible to pass garbage to realloc, but because its definition is so general it's hard to guard against invalid arguments. If you pass a NULL pointer by mistake, that's legal. If you pass a 0 size by mistake, that's legal too. It's too bad if you malloc a new block or free the current one when your intent is to resize a block. How can you assert that realloc's arguments are valid if practically everything is legal? No matter what you throw at it, realloc handles it, even to extremes. At one extreme it frees blocks; at the other it mallocs them. These are totally opposite behaviors.

=====

[Then goes on further to discuss safe and solid memory management. As mentioned in the book, Maguire took over (or managed) a particular group of new programmers that were very unaware of dangrous coding techniques. The impetus was, as he states, that Microsoft was forced to cancel a (probably important) unannounced product because of a run-away bug list. Not a new phenomom in the industry but it was the first time it had happened to Microsoft and on a major project. But counts did fall to their previous low levels but apparently this product was never released.

Anyway, the book is certainly worth reading even though there's no C++ or anything fancy in it. It's just straight C and common sense and logic.

alancarre@xxxxxxxxxxxx  
– Alan Carre

Thank you very much,I will use a map to record the size of those memory blocks.My policy is wrong.

Re: How to get the memory block size which allocated by new operator?

Re: How to get the memory block size which allocated by new operator?