Will this C++ code cause a memory leak (casting array new)

Asked 16 years, 3 months ago Modified 4 years, 9 months ago Viewed 8k times

10

I have been working on some legacy C++ code that uses variable length structures (TAPI), where the structure size will depend on variable length strings. The structures are allocated by casting array `new` thus:

```
STRUCT* pStruct = (STRUCT*)new BYTE[sizeof(STRUCT) + nPaddingSize];
```

Later on however the memory is freed using a `delete` call:

```
delete pStruct;
```

Will this mix of array `new[]` and non-array `delete` cause a memory leak or would it depend on the compiler? Would I be better off changing this code to use `malloc` and `free` instead?

c++ memory-management memory-leaks

Share
Improve this question
Follow

edited Mar 23, 2020 at 12:38

MAChitgarha
4,258 ● 2 ● 37 ● 42

asked Sep 16, 2008 at 14:47

Rob
78.5k ● 57 ● 161 ● 199

A very similar question has already been asked. See <stackoverflow.com/questions/70880/… > – ChrisN Sep 16, 2008 at 14:55

24 Answers

Sorted by: Highest score (default)

12

Technically I believe it could cause a problem with mismatched allocators, though in practice I don't know of any compiler that would not do the right thing with this example.

More importantly if `STRUCT` where to have (or ever be given) a destructor then it would invoke the destructor without having invoked the corresponding constructor.

Of course, if you know where pStruct came from why not just cast it on delete to match the allocation:

```
delete [] (BYTE*) pStruct;
```

Share

Improve this answer

Follow

edited Dec 22, 2015 at 22:52

LogicStuff

19.6k ● 6 ● 56 ● 74

answered Sep 16, 2008 at 14:50

Rob Walker

47.4k ● 15 ● 100 ● 137

In this case you need to explicitly call STRUCT's destructor before doing the delete, as in Richard Corden's answer. – Patrick Johnmeyer Sep 16, 2008 at 17:12

I don't see how "I don't know of any compiler that would not do the right thing" could be accurate in light of what we know about Visual C++ due to Raymond Chen and other sources (see link in Len Holgate's answer). – Adam Mitz Sep 17, 2008 at 0:42

7

I personally think you'd be better off using `std::vector` to manage your memory, so you don't need the `delete`.

```
std::vector<BYTE> backing(sizeof(STRUCT) + nPaddingSize);
STRUCT* pStruct = (STRUCT*)(&backing[0]);
```

Once backing leaves scope, your `pStruct` is no longer valid.

Or, you can use:

```
boost::scoped_array<BYTE> backing(new BYTE[sizeof(STRUCT) + nPaddingSize]);
STRUCT* pStruct = (STRUCT*)backing.get();
```

Or `boost::shared_array` if you need to move ownership around.

Share

Improve this answer

Follow

edited Dec 22, 2015 at 22:52

LogicStuff

19.6k ● 6 ● 56 ● 74

answered Sep 16, 2008 at 15:39

Matt Cruikshank

2,948 ● 21 ● 24

6

Yes it will cause a memory leak.

See this except from C++ Gotchas: http://www.informit.com/articles/article.aspx?p=30642 for why.

Raymond Chen has an explanation of how vector `new` and `delete` differ from the scalar versions under the covers for the Microsoft compiler... Here: http://blogs.msdn.com/oldnewthing/archive/2004/02/03/66660.aspx

IMHO you should fix the delete to:

```
delete [] pStruct;
```

rather than switching to `malloc` / `free`, if only because it's a simpler change to make without making mistakes ;)

And, of course, the simpler to make change that I show above is wrong due to the casting in the original allocation, it should be

```
delete [] reinterpret_cast<BYTE *>(pStruct);
```

so, I guess it's probably as easy to switch to `malloc` / `free` after all ;)

Share
Improve this answer
Follow

edited Dec 22, 2015 at 22:53

LogicStuff
19.6k ● 6 ● 56 ● 74

answered Sep 16, 2008 at 15:00

Len Holgate
21.6k ● 4 ● 47 ● 94

6

The behaviour of the code is undefined. You may be lucky (or not) and it may work with your compiler, but really that's not correct code. There's two problems with it:

1. The `delete` should be an array `delete []`.
2. The `delete` should be called on a pointer to the same type as the type allocated.

So to be entirely correct, you want to be doing something like this:

```
delete [] (BYTE*)(pStruct);
```

Share
Improve this answer
Follow

edited Dec 22, 2015 at 22:55

LogicStuff
19.6k ● 6 ● 56 ● 74

answered Sep 16, 2008 at 14:52

slicedlime
2,152 ● 1 ● 17 ● 16

4

The C++ standard clearly states:

```
delete-expression:
            ::opt delete cast-expression
            ::opt delete [ ] cast-expression
```

> The first alternative is for non-array objects, and the second is for arrays. The operand shall have a pointer type, or a class type having a single conversion

> function (12.3.2) to a pointer type. The result has type void.
>
> In the first alternative (delete object), the value of the operand of delete shall be a pointer to a non-array object [...] If not, the behavior is undefined.

The value of the operand in `delete pStruct` is a pointer to an array of `char`, independent of its static type (`STRUCT*`). Therefore, any discussion of memory leaks is quite pointless, because the code is ill-formed, and a C++ compiler is not required to produce a sensible executable in this case.

It could leak memory, it could not, or it could do anything up to crashing your system. Indeed, a C++ implementation with which I tested your code aborts the program execution at the point of the delete expression.

Share Improve this answer Follow

answered Sep 16, 2008 at 15:14

ben

2,014 ● 14 ● 20

3

As highlighted in other posts:

1) Calls to new/delete allocate memory and may call constructors/destructors (C++ '03 5.3.4/5.3.5)

2) Mixing array/non-array versions of `new` and `delete` is undefined behaviour. (C++ '03 5.3.5/4)

Looking at the source it appears that someone did a search and replace for `malloc` and `free` and the above is the result. C++ does have a direct replacement for these functions, and that is to call the allocation functions for `new` and `delete` directly:

```
STRUCT* pStruct = (STRUCT*)::operator new (sizeof(STRUCT) + nPaddingSize);
// ...
pStruct->~STRUCT ();  // Call STRUCT destructor
::operator delete (pStruct);
```

If the constructor for STRUCT should be called, then you could consider allocating the memory and then use placement `new`:

```
BYTE * pByteData = new BYTE[sizeof(STRUCT) + nPaddingSize];
STRUCT * pStruct = new (pByteData) STRUCT ();
// ...
pStruct->~STRUCT ();
delete[] pByteData;
```

Share

Improve this answer

Follow

edited Dec 22, 2015 at 22:56

LogicStuff

19.6k ● 6 ● 56 ● 74

answered Sep 16, 2008 at 16:27

Richard Corden

21.7k ● 9 ● 61 ● 87

2

@eric - Thanks for the comments. You keep saying something though, that drives me nuts:

> Those run-time libraries handle the memory management calls to the OS in a OS independent consistent syntax and those run-time libraries are responsible for making malloc and new work consistently between OSes such as Linux, Windows, Solaris, AIX, etc....

This is not true. The compiler writer provides the implementation of the std libraries, for instance, and they are absolutely free to implement those in an OS **dependent** way. They're free, for instance, to make one giant call to malloc, and then manage memory within the block however they wish.

Compatibility is provided because the API of std, etc. is the same - not because the run-time libraries all turn around and call the exact same OS calls.

Share Improve this answer Follow

answered Sep 17, 2008 at 16:27

Matt Cruikshank

2,948 ● 21 ● 24

2

The various possible uses of the keywords new and delete seem to create a fair amount of confusion. There are always two stages to constructing dynamic objects in C++: the allocation of the raw memory and the construction of the new object in the allocated memory area. On the other side of the object lifetime there is the destruction of the object and the deallocation of the memory location where the object resided.

Frequently these two steps are performed by a single C++ statement.

```
MyObject* ObjPtr = new MyObject;

//...

delete MyObject;
```

Instead of the above you can use the C++ raw memory allocation functions `operator new` and `operator delete` and explicit construction (via placement `new`) and destruction to perform the equivalent steps.

```
void* MemoryPtr = ::operator new( sizeof(MyObject) );
MyObject* ObjPtr = new (MemoryPtr) MyObject;

// ...

ObjPtr->~MyObject();
::operator delete( MemoryPtr );
```

Notice how there is no casting involved, and only one type of object is constructed in the allocated memory area. Using something like `new char[N]` as a way to allocate raw memory is technically incorrect as, logically, `char` objects are created in the newly allocated memory. I don't know of any situation where it doesn't 'just work' but it blurs the distinction between raw memory allocation and object creation so I advise against it.

In this particular case, there is no gain to be had by separating out the two steps of `delete` but you do need to manually control the initial allocation. The above code works in the 'everything working' scenario but it will leak the raw memory in the case where the constructor of `MyObject` throws an exception. While this could be caught and solved with an exception handler at the point of allocation it is probably neater to provide a custom operator new so that the complete construction can be handled by a placement new expression.

```
class MyObject
{
    void* operator new( std::size_t rqsize, std::size_t padding )
    {
        return ::operator new( rqsize + padding );
    }

    // Usual (non-placement) delete
    // We need to define this as our placement operator delete
    // function happens to have one of the allowed signatures for
    // a non-placement operator delete
    void operator delete( void* p )
    {
        ::operator delete( p );
    }

    // Placement operator delete
    void operator delete( void* p, std::size_t )
    {
        ::operator delete( p );
    }
};
```

There are a couple of subtle points here. We define a class placement new so that we can allocate enough memory for the class instance plus some user specifiable padding. Because we do this we need to provide a matching placement delete so that if the memory allocation succeeds but the construction fails, the allocated memory is

automatically deallocated. Unfortunately, the signature for our placement delete matches one of the two allowed signatures for non-placement delete so we need to provide the other form of non-placement delete so that our real placement delete is treated as a placement delete. (We could have got around this by adding an extra dummy parameter to both our placement new and placement delete, but this would have required extra work at all the calling sites.)

```
// Called in one step like so:
MyObject* ObjectPtr = new (padding) MyObject;
```

Using a single new expression we are now guaranteed that memory won't leak if any part of the new expression throws.

At the other end of the object lifetime, because we defined operator delete (even if we hadn't, the memory for the object originally came from global operator new in any case), the following is the correct way to destroy the dynamically created object.

```
delete ObjectPtr;
```

Summary!

1. Look no casts! `operator new` and `operator delete` deal with raw memory, placement new can construct objects in raw memory. An explicit cast from a `void*` to an object pointer is usually a sign of something logically wrong, even if it does 'just work'.
2. We've completely ignored new[] and delete[]. These variable size objects will not work in arrays in any case.
3. Placement new allows a new expression not to leak, the new expression still evaluates to a pointer to an object that needs destroying and memory that needs deallocating. Use of some type of smart pointer may help prevent other types of leak. On the plus side we've let a plain `delete` be the correct way to do this so most standard smart pointers will work.

Share

Improve this answer

Follow

edited Oct 8, 2008 at 21:58

answered Sep 20, 2008 at 15:19

CB Bailey

789k ● 107 ● 643 ● 664

2

If you *really* must do this sort of thing, you should probably call operator `new` directly:

```
STRUCT* pStruct = operator new(sizeof(STRUCT) + nPaddingSize);
```

I believe calling it this way avoids calling constructors/destructors.

Share

Improve this answer

Follow

edited Dec 22, 2015 at 22:57

LogicStuff

19.6k • 6 • 56 • 74

answered Sep 16, 2008 at 16:20

Ferruccio

101k • 38 • 229 • 303

The result of operator new needs static_cast-ing to the appropriate type – James Hopkin Sep 16, 2008 at 16:41

This has the advantage over malloc/free: if the global operator new has been replaced, this will use it. – James Hopkin Sep 16, 2008 at 16:42

But this won't work if STRUCT has overridden operator new, will it? I would prefer placement new syntax here; it will call the "right" placement new per type. – Patrick Johnmeyer Sep 16, 2008 at 17:17

To be clear, when freeing the memory, all you need to do is "delete pStruct;" right? – Greg Rogers Oct 8, 2008 at 22:12

1

I am currently unable to vote, but slicedlime's answer is preferable to Rob Walker's answer, since the problem has nothing to do with allocators or whether or not the STRUCT has a destructor.

Also note that the example code does not necessarily result in a memory leak - it's undefined behavior. Pretty much anything could happen (from nothing bad to a crash far, far away).

The example code results in undefined behavior, plain and simple. slicedlime's answer is direct and to the point (with the caveat that the word 'vector' should be changed to 'array' since vectors are an STL thing).

This kind of stuff is covered pretty well in the C++ FAQ (Sections 16.12, 16.13, and 16.14):

http://www.parashift.com/c++-faq-lite/freestore-mgmt.html#faq-16.12

Share

Improve this answer

Follow

edited May 23, 2017 at 12:33

Community Bot

1 • 1

answered Sep 16, 2008 at 16:27

Michael Burr

340k • 52 • 548 • 769

1

It's an array delete ([]) you're referring to, not a vector delete. A vector is std::vector, and it takes care of deletion of its elements.

Share Improve this answer Follow

answered Sep 16, 2008 at 16:47

Assaf Lavie
75.7k • 35 • 150 • 205

0

You'd could cast back to a BYTE * and the delete:

```
delete[] (BYTE*)pStruct;
```

Share Improve this answer Follow

answered Sep 16, 2008 at 14:52

Eclipse
45.5k • 20 • 116 • 172

0

Yes that may, since your allocating with new[] but deallocating with delelte, yes malloc/free is safer here, but in c++ you should not use them since they won't handle (de)constructors.

Also your code will call the deconstructor, but not the constructor. For some structs this may cause a memory leak (if the constructor allocated further memory, eg for a string)

Better would be to do it correctly, as this will also correctly call any constructors and deconstructors

```
STRUCT* pStruct = new STRUCT;
...
delete pStruct;
```

Share

Improve this answer

Follow

edited Sep 16, 2008 at 14:55

answered Sep 16, 2008 at 14:49

Fire Lancer
30.1k • 33 • 122 • 184

Note: this does not address the original purpose of STRUCT, in that it is a variable sized structure. It may, for instance, have a variable sized array as its last member, or it may require additional space after the end of the struct as a matter of convention. – Aaron Sep 17, 2008 at 0:10

It's always best to keep acquisition/release of any resource as balanced as possible. Although leaking or not is hard to say in this case. It depends on the compiler's implementation of the vector (de)allocation.

0

```
BYTE * pBytes = new BYTE [sizeof(STRUCT) + nPaddingSize];

STRUCT* pStruct = reinterpret_cast< STRUCT* > ( pBytes ) ;

 // do stuff with pStruct

delete [] pBytes ;
```

Share

Improve this answer

Follow

edited Sep 16, 2008 at 15:00

answered Sep 16, 2008 at 14:54

QBziZ

3,250 ● 26 ● 24

0

Len: the problem with that is that pStruct is a STRUCT*, but the memory allocated is actually a BYTE[] of some unknown size. So delete[] pStruct will not de-allocate all of the allocated memory.

Share Improve this answer Follow

answered Sep 16, 2008 at 15:04

ReedHedges

0

You're sort of mixing C and C++ ways of doing things. Why allocate more than the size of a STRUCT? Why not just "new STRUCT"? If you must do this then it might be clearer to use malloc and free in this case, since then you or other programmers might be a little less likely to make assumptions about the types and sizes of the allocated objects.

Share

Improve this answer

Follow

edited Sep 16, 2008 at 15:05

answered Sep 16, 2008 at 14:55

ReedHedges

> @Reed: because this may be a variable length struct, or a struct that, by convention, has some ammount of buffer space following it for specific purposes. For example, a data structure holding strings may place those strings in the same heap allocation. new BYTE[sizeof(STRUCT) + stringSpace]. – Aaron Sep 17, 2008 at 0:18

0

@Matt Cruikshank You should pay attention and read what I wrote again because I never suggested not calling delete[] and just let the OS clean up. And you're wrong about the C++ run-time libraries managing the heap. If that were the case then C++ would not be portable as is today and a crashing application would never get cleaned up by the OS. (acknowledging there are OS specific run-times that make C/C++

appear non-portable). I challenge you to find stdlib.h in the Linux sources from kernel.org. The new keyword in C++ actually is talking to the same memory management routines as malloc.

The C++ run-time libraries make OS system calls and it's the OS that manages the heaps. You are partly correct in that the run-time libraries indicate when to release the memory however, they don't actually walk any heap tables directly. In other words, the runtime you link against does not add code to your application to walk heaps to allocate or deallocate. This is the case in Windows, Linux, Solaris, AIX, etc... It's also the reason you won't fine malloc in any Linux's kernel source nor will you find stdlib.h in Linux source. Understand these modern operating system have virtual memory managers that complicates things a bit further.

Ever wonder why you can make a call to malloc for 2G of RAM on a 1G box and still get back a valid memory pointer?

Memory management on x86 processors is managed within Kernel space using three tables. PAM (Page Allocation Table), PD (Page Directories) and PT (Page Tables). This is at the hardware level I'm speaking of. One of the things the OS memory manager does, not your C++ application, is to find out how much physical memory is installed on the box during boot with help of BIOS calls. The OS also handles exceptions such as when you try to access memory your application does not have rights too. (GPF General Protection Fault).

It may be that we are saying the same thing Matt, but I think you may be confusing the under hood functionality a bit. I use to maintain a C/C++ compiler for a living...

Share Improve this answer Follow

answered Sep 17, 2008 at 3:23

Eric

1,697 ● 6 ● 29 ● 39

0

@ericmayo - cripes. Well, experimenting with VS2005, I can't get an honest leak out of scalar delete on memory that was made by vector new. I guess the compiler behavior is "undefined" here, is about the best defense I can muster.

You've got to admit though, it's a really lousy practice to do what the original poster said.

> If that were the case then C++ would not be portable as is today and a crashing application would never get cleaned up by the OS.

This logic doesn't really hold, though. My assertion is that a compiler's runtime can manage the memory within the memory blocks that the OS returns to it. This is how

most virtual machines work, so your argument against portability in this case don't make much sense.

Share Improve this answer Follow

answered Sep 17, 2008 at 14:42

Matt Cruikshank
2,948 ● 21 ● 24

0

@Matt Cruikshank

"Well, experimenting with VS2005, I can't get an honest leak out of scalar delete on memory that was made by vector new. I guess the compiler behavior is "undefined" here, is about the best defense I can muster."

I disagree that it's a compiler behavior or even a compiler issue. The 'new' keyword gets compiled and linked, as you pointed out, to run-time libraries. Those run-time libraries handle the memory management calls to the OS in a OS independent consistent syntax and those run-time libraries are responsible for making malloc and new work consistently between OSes such as Linux, Windows, Solaris, AIX, etc.... This is the reason I mentioned the portability argument; an attempt to prove to you that the run-time does not actually manage memory either.

The OS manages memory.

The run-time libs interface to the OS.. On Windows, this is the virtual memory manager DLLs. This is why stdlib.h is implemented within the GLIB-C libraries and not the Linux kernel source; if GLIB-C is used on other OSes, it's implementation of malloc changes to make the correct OS calls. In VS, Borland, etc.. you will never find any libraries that ship with their compilers that actually manage memory either. You will, however, find OS specific definitions for malloc.

Since we have the source to Linux, you can go look at how malloc is implemented there. You will see that malloc is actually implemented in the GCC compiler which, in turn, basically makes two Linux system calls into the kernel to allocate memory. Never, malloc itself, actually managing memory!

And don't take it from me. Read the source code to Linux OS or you can see what K&R say about it... Here is a PDF link to the K&R on C.

http://www.oberon2005.ru/paper/kr_c.pdf

See near end of Page 149: "Calls to malloc and free may occur in any order; malloc calls upon the operating system to obtain more memory as necessary. These routines illustrate some of the considerations involved in writing machine-dependent code in a

relatively machineindependent way, and also show a real-life application of structures, unions and typedef."

"You've got to admit though, it's a really lousy practice to do what the original poster said."

Oh, I don't disagree there. My point was that the original poster's code was not conducive of a memory leak. That's all I was saying. I didn't chime in on the best practice side of things. Since the code is calling delete, the memory is getting free up.

I agree, in your defense, if the original poster's code never exited or never made it to the delete call, that the code could have a memory leak but since he states that later on he sees the delete getting called. "Later on however the memory is freed using a delete call:"

Moreover, my reason for responding as I did was due to the OP's comment "variable length structures (TAPI), where the structure size will depend on variable length strings"

That comment sounded like he was questioning the dynamic nature of the allocations against the cast being made and was consequentially wondering if that would cause a memory leak. I was reading between the lines if you will ;).

Share Improve this answer Follow

answered Sep 17, 2008 at 15:16

Eric

1,697 ● 6 ● 29 ● 39

> The problem (or rather 'a' problem) is that the allocation is done using the array new operator and the delete is done using non-array delete. This is undefined behavior. Some compilers (including one I'm using) keep housekeeping info for array new's that will not be freed with a non-array delete. – Michael Burr Sep 30, 2008 at 6:45

0

In addition to the excellent answers above, I would also like to add:

If your code runs on linux or if you can compile it on linux then I would suggest running it through Valgrind. It is an excellent tool, among the myriad of useful warnings it produces it also will tell you when you allocate memory as an array and then free it as a non-array (and vice-versa).

Share Improve this answer Follow

answered Sep 17, 2008 at 15:29

KPexEA

16.8k ● 16 ● 63 ● 79

Use operator new and delete:

```
struct STRUCT
{
  void *operator new (size_t)
  {
    return new char [sizeof(STRUCT) + nPaddingSize];
  }

  void operator delete (void *memory)
  {
    delete [] reinterpret_cast <char *> (memory);
  }
};

void main()
{
  STRUCT *s = new STRUCT;
  delete s;
}
```

Share

Improve this answer

Follow

edited Jul 3, 2012 at 13:23

user142162

answered Sep 16, 2008 at 15:07

Skizz

71k • 10 • 74 • 109

Note: nPaddingSize may be dynamic, in which case operator new should take two parameters, (size_t classSize, size_t padding). Also: return new char[classSize+padding] s = new (nPaddingSize) STRUCT Note: if providing one allocator, provide all 8 default 'flavors'. make private if not desired. – Aaron Sep 17, 2008 at 0:23

I think the is no memory leak.

```
STRUCT* pStruct = (STRUCT*)new BYTE [sizeof(STRUCT) + nPaddingSize];
```

This gets translated into a memory allocation call within the operating system upon which a pointer to that memory is returned. At the time memory is allocated, the size of `sizeof(STRUCT)` and the size of `nPaddingSize` would be known in order to fulfill any memory allocation requests against the underlying operating system.

So the memory that is allocated is "recorded" in the operating system's global memory allocation tables. Memory tables are indexed by their pointers. So in the corresponding call to delete, all memory that was originally allocated is free. (memory fragmentation a popular subject in this realm as well).

You see, the C/C++ compiler is not managing memory, the underlying operating system is.

I agree there are cleaner methods but the OP did say this was legacy code.

In short, I don't see a memory leak as the accepted answer believes there to be one.

Share
Improve this answer
Follow

edited Jul 3, 2012 at 13:23
user142162

answered Sep 16, 2008 at 16:42
Eric
1,697 ● 6 ● 29 ● 39

-1

Rob Walker reply is good.

Just small addition, if you don't have any constructor or/and distructors, so you basically need allocate and free a chunk of raw memory, consider using free/malloc pair.

Share
Improve this answer
Follow

edited May 23, 2017 at 11:59
Community Bot
1 ● 1

answered Sep 16, 2008 at 14:54
Serge
7,694 ● 5 ● 41 ● 46

Note: adding malloc/free to a code base opens up countless new oppertunities to mis-match allocators. – Aaron Sep 17, 2008 at 0:14

I don't want to start arguing but, could you please give an example how malloc/free will cause allocators mismatch or some reference. – Serge Sep 17, 2008 at 7:34

-1

ericmayo.myopenid.com is so wrong, that someone with enough reputation should downvote him.

The C or C++ runtime libraries are managing the heap which is given to it in blocks by the Operating System, somewhat like you indicate, Eric. But it *is* the responsibility of the developer to indicate to the compiler which runtime calls should be made to free memory, and possibly destruct the objects that are there. Vector delete (aka delete[]) is necessary in this case, in order for the C++ runtime to leave the heap in a valid state. The fact that when the PROCESS terminates, the OS is smart enough to deallocate the underlying memory blocks is not something that developers should rely on. This would be like never calling delete at all.

Share Improve this answer Follow

answered Sep 16, 2008 at 19:20
Matt Cruikshank
2,948 ● 21 ● 24