Can placement new for arrays be used in a portable way?

Asked 16 years, 4 months ago Modified 3 months ago Viewed 19k times

50

Is it possible to actually make use of placement new in portable code when using it for arrays?

It appears that the pointer you get back from new[] is not always the same as the address you pass in (5.3.4, note 12 in the standard seems to confirm that this is correct), but I don't see how you can allocate a buffer for the array to go in if this is the case.

The following example shows the problem. Compiled with Visual Studio, this example results in memory corruption:

```
#include <new>
#include <stdio.h>

class A
{
    public:

    A() : data(0) {}
    virtual ~A() {}
    int data;
};

int main()
{
    const int NUMELEMENTS=20;

    char *pBuffer = new char[NUMELEMENTS*sizeof(A)];
    A *pA = new(pBuffer) A[NUMELEMENTS];

    // With VC++, pA will be four bytes higher than pBuffer
    printf("Buffer address: %x, Array address: %x\n", pBuffer, pA);

    // Debug runtime will assert here due to heap corruption
    delete[] pBuffer;

    return 0;
}
```

Looking at the memory, the compiler seems to be using the first four bytes of the buffer to store a count of the number of items in it. This means that because the buffer is only `sizeof(A)*NUMELEMENTS` big, the last element in the array is written into unallocated heap.

So the question is can you find out how much additional overhead your implementation wants in order to use placement new[] safely? Ideally, I need a technique that's portable between different compilers. Note that, at least in VC's case, the overhead seems to differ for different classes. For instance, if I remove the virtual destructor in the example, the address returned from new[] is the same as the address I pass in.

c++ arrays portability overhead

Share
Improve this question
Follow

edited Aug 27 at 12:43

Jonas
128k ● 100 ● 326 ● 405

asked Aug 18, 2008 at 21:33

James Sutherland
3,863 ● 2 ● 28 ● 24

1 Ah curses. I made a dupe of your question :(Array placement-new requires unspecified overhead in the buffer? – Mooing Duck May 7, 2015 at 18:51

1 Hmm... if the overhead disappears when you remove the virtual destructor, that would suggest that the overhead is likely from either the class' vtable, or from VStudio's implementation of RTTI. – Justin Time - Reinstate Monica May 25, 2016 at 18:33

1 Or at the very least, part of the overhead is. It's also possible that overhead is only used if the class has a non-trivial destructor. – Justin Time - Reinstate Monica May 25, 2016 at 18:50

8 Answers

Sorted by: Highest score (default)

34

Personally I'd go with the option of not using placement new on the array and instead use placement new on each item in the array individually. For example:

```
int main(int argc, char* argv[])
{
  const int NUMELEMENTS=20;

  char *pBuffer = new char[NUMELEMENTS*sizeof(A)];
  A *pA = (A*)pBuffer;

  for(int i = 0; i < NUMELEMENTS; ++i)
  {
    pA[i] = new (pA + i) A();
  }

  printf("Buffer address: %x, Array address: %x\n", pBuffer, pA);

  // dont forget to destroy!
  for(int i = 0; i < NUMELEMENTS; ++i)
  {
    pA[i].~A();
  }
```

```
    delete[] pBuffer;

    return 0;
}
```

Regardless of the method you use, make sure you manually destroy each of those items in the array before you delete pBuffer, as you could end up with leaks ;)

Note: I haven't compiled this, but I think it should work (I'm on a machine that doesn't have a C++ compiler installed). It still indicates the point :) Hope it helps in some way!

Edit:

The reason it needs to keep track of the number of elements is so that it can iterate through them when you call delete on the array and make sure the destructors are called on each of the objects. If it doesn't know how many there are it wouldn't be able to do this.

Share

Improve this answer

Follow

edited Mar 13, 2015 at 6:52

Criticize SE actions means ban

58.8k ● 7 ● 53 ● 99

answered Aug 18, 2008 at 22:53

OJ.

29.4k ● 6 ● 58 ● 71

1 VC++ is a bad compiler. The default placement new array aka `new(void*) type[n]` performs in-place construction of `n` objects of `type`. The provided pointer has to be properly aligned to match `alignof(type)` (note: `sizeof(type)` is a multiple of `alignof(type)` due to padding). Because you would normally have to carry around the array's length, there is no actual requirement to store it inside the array, because you're going to destroy it with a for-loop anyways (no placement delete operators). – bit2shift Mar 11, 2016 at 21:45

2 @bit2shift The C++ standard explicitly states that `new[]` pads the allocated memory, although it allows a padding of 0 bytes. (See section "expr.new", specifically examples (14.3) and (14.4) and the explanation beneath them.) So, in that regard, it's actually standard-compliant. [If you don't have a copy of the final version of the C++14 standard, see page 133 here.] – Justin Time - Reinstate Monica May 25, 2016 at 18:44

2 @JustinTime That's a defect no one seems to bat an eye at. – bit2shift May 26, 2016 at 21:41

5 @JustinTime It's a defect in the standard itself to allow an overhead to be applied to `void* operator new[](std::size_t count, void* ptr);` when this operator is known to be "no-op" (no allocation) and it's also clearly known that a pointer returned by this operator or its scalar sibling cannot be passed to either `delete` or `delete[]`, requiring the programmer to manually destroy each element. – bit2shift May 29, 2016 at 2:29

2 With regard to treating a collection of adjacent objects individually constructed in contiguous memory as an array object, see C++ Core Issue 2182. Unfortunately this doesn't really seem to have been resolved yet, which means there might not be any way to in-place construct an array object that's guaranteed to work by the standard without any undefined behaviour. This problem is also analyzed in this video. – Matthijs Oct 3, 2019 at 23:00

6

@Derek

5.3.4, section 12 talks about the array allocation overhead and, unless I'm misreading it, it seems to suggest to me that it is valid for the compiler to add it on placement new as well:

> This overhead may be applied in all array new-expressions, including those referencing the library function operator new[](std::size_t, void*) and other placement allocation functions. The amount of overhead may vary from one invocation of new to another.

That said, I think VC was the only compiler that gave me trouble with this, out of it, GCC, Codewarrior and ProDG. I'd have to check again to be sure, though.

Share Improve this answer Follow

answered Aug 19, 2008 at 10:16

James Sutherland

3,863 ● 2 ● 28 ● 24

1 I would be shocked if VC was the only compiler adding extra space, I would think all compilers would store the number of destructors to call there. There's no other logical place to put it. – Mooing Duck May 7, 2015 at 19:02

3 @MooingDuck there's no placement delete operator to make use of the aforementioned "array length". In fact, you have to manually call the destructors for an array **constructed** with `new(pointer) type[length]`. VC is a bad compiler, everyone should know that. – bit2shift Mar 11, 2016 at 21:23

4

@James

> I'm not even really clear why it needs the additional data, as you wouldn't call delete[] on the array anyway, so I don't entirely see why it needs to know how many items are in it.

After giving this some thought, I agree with you. There is no reason why placement new should need to store the number of elements, because there is no placement delete. Since there's no placement delete, there's no reason for placement new to store the number of elements.

I also tested this with gcc on my Mac, using a class with a destructor. On my system, placement new was *not* changing the pointer. This makes me wonder if this is a VC++ issue, and whether this might violate the standard (the standard doesn't specifically address this, so far as I can find).

Share Improve this answer Follow

answered Aug 19, 2008 at 2:14

Derek Park

46.8k ● 16 ● 59 ● 76

I tested with both clang 3.7.0 and GCC 5.3.0, both with `-std=c++14` and `-pedantic` on Coliru. Neither showed the existence of an overhead, specially with classes with nontrivial destructors. So, I'm thinking, this is another example of how Visual C++ is such a bad compiler. – bit2shift Mar 11, 2016 at 21:17

I very much doubt that, @Nik-Lz, considering they still flaunt the hideous `nothrownew.obj` as a way to make `new` behave like `new(std::nothrow)`. – bit2shift Sep 5, 2018 at 4:05

3

Thanks for the replies. Using placement new for each item in the array was the solution I ended up using when I ran into this (sorry, should have mentioned that in the question). I just felt that there must have been something I was missing about doing it with placement new[]. As it is, it seems like placement new[] is essentially unusable thanks to the standard allowing the compiler to add an additional

unspecified overhead to the array. I don't see how you could ever use it safely and portably.

I'm not even really clear why it needs the additional data, as you wouldn't call delete[] on the array anyway, so I don't entirely see why it needs to know how many items are in it.

Share Improve this answer Follow

answered Aug 19, 2008 at 0:03

James Sutherland
3,863 ● 2 ● 28 ● 24

3

Placement new itself is portable, but the assumptions you make about what it does with a specified block of memory are not portable. Like what was said before, if you were a compiler and were given a chunk of memory, how would you know how to allocate an array and properly destruct each element if all you had was a pointer? (See the interface of operator delete[].)

Edit:

And there actually is a placement delete, only it is only called when a constructor throws an exception while allocating an array with placement new[].

Whether new[] actually needs to keep track of the number of elements somehow is something that is left up to the standard, which leaves it up to the compiler. Unfortunately, in this case.

Share

Improve this answer

Follow

edited Jul 3, 2012 at 14:51

user142162

answered Aug 19, 2008 at 21:36

Mat Noguchi
1,080 ● 6 ● 7

1 How can it ever be "portable" if it overwrites an arbitrarily large amount of storage? You could never safely call it since you never know how much it is going to write. – BeeOnRope Oct 19, 2017 at 22:18

1 The `delete[]` expression needs to know, but it can only be used on the result of a `new[]` expression with heap allocation (either throwing or `nothrow`). For placement `new[]` it is really unnecessary. The only explanation I have for VC's behavior is that heap `new[]` is somehow implemented in terms of placement `new[]` and doesn't store the number of elements itself. – Arne Vogel Jan 9, 2018 at 18:40

3

C++17 (draft N4659) says in [expr.new], paragraph 15:

> [O]verhead may be applied in all array *new-expressions*, including those referencing the library function `operator new[](std::size_t, void*)` and

> other placement allocation functions. The amount of overhead may vary from one invocation of `new` to another.

So it appears to be impossible to use `(void*)` placement `new[]` safely in C++17 (and earlier), and it's unclear to me why it's even specified to exist.

In C++20 (draft N4861) this was changed to

> [O]verhead may be applied in all array *new-expressions*, including those referencing a placement allocation function, except when referencing the library function `operator new[](std::size_t, void*)`. The amount of overhead may vary from one invocation of `new` to another.

So if you're sure that you're using C++20, you can safely use it—but only that one placement form, and only (it appears) if you don't override the standard definition.

Even the C++20 text seems ridiculous, because the only purpose of the extra space is to store array-size metadata, but there is no way to access it when using any custom placement form of `new[]`. It's in a private format that only `delete[]` knows how to read—and with custom allocation you can't use `delete[]`, so at best it's just wasted space.

Actually, as far as I can tell, there is no safe way to use custom forms of `operator new[]` at all. There is no way to call the destructors correctly because the necessary information isn't passed to `operator new[]`. Even if you know that the objects are trivially destructible, the `new` expression may return a pointer to some arbitrary location in the middle of the memory block that your `operator new[]` returned (skipping over the pointless metadata), so you can't wrap an allocation library that only supplies `malloc` and `free` equivalents: it also needs a way to search for a block by a pointer to its middle, which even if it exists is likely to be a lot slower.

I don't understand how they (or just Stroustrup?) botched this so badly. The obviously correct way to do it is to pass the number of array elements and the size of each element to `operator new[]` as two arguments, and let each allocator choose how to store it. Perhaps I'm missing something.

Share Improve this answer Follow

answered Feb 11, 2023 at 7:01

benrg

1,845 ● 17 ● 15

> You can't override placement-new, me thinks? As long as you call it as `::new((void*)addr) type` (which is what `std::construct_at` does). – HolyBlackCat Feb 11, 2023 at 8:02

At least in the latest draft, allocation overhead is forbidden for all non-allocating forms of new: eel.is/c++draft/expr.new#16 – Blaisorblade Dec 15, 2023 at 5:58

2

Similar to how you would use a single element to calculate the size for one placement-new, use an array of those elements to calculate the size required for an array.

If you require the size for other calculations where the number of elements may not be known you can use sizeof(A[1]) and multiply by your required element count.

e.g

```
char *pBuffer = new char[ sizeof(A[NUMELEMENTS]) ];
A *pA = (A*)pBuffer;

for(int i = 0; i < NUMELEMENTS; ++i)
{
    pA[i] = new (pA + i) A();
}
```

Share Improve this answer Follow

answered Aug 18, 2008 at 23:26

Andrew Grant
58.8k ● 22 ● 131 ● 144

2 The point is that MSVC apparently requires additional space beyond the value of `sizeof(A[NUMELEMENTS])` in the case of `new []`. `sizeof(A[N])` is just going to be `N * sizeof(N)` in general and won't reflect this additional required space. – BeeOnRope Oct 9, 2017 at 7:53

-1

I think gcc does the same thing as MSVC, but of course this doesn't make it "portable".

I think you can work around the problem when NUMELEMENTS is indeed a compile time constant, like so:

```
typedef A Arr[NUMELEMENTS];

A* p = new (buffer) Arr;
```

This should use the scalar placement new.

Share Improve this answer Follow

answered Aug 18, 2008 at 21:45

Yossi Kreinin

246 ● 1 ● 3

6 It doesn't. `operator new()` vs `operator new[]()` depends on whether there is an array type involved, not whether there were `[]` in the source code. – Ben Voigt Jun 27, 2014 at 15:08