Why do we even need the "delete[]" operator?

Asked 16 years, 1 month ago Modified 4 years, 6 months ago Viewed 8k times

43

This is a question that's been nagging me for some time. I always thought that C++ should have been designed so that the `delete` operator (without brackets) works even with the `new[]` operator.

In my opinion, writing this:

```
int* p = new int;
```

should be equivalent to allocating an array of 1 element:

```
int* p = new int[1];
```

If this was true, the `delete` operator could always be deleting arrays, and we wouldn't need the `delete[]` operator.

Is there any reason why the `delete[]` operator was introduced in C++? The only reason I can think of is that allocating arrays has a small memory footprint (you have to store the array size somewhere), so that distinguishing `delete` vs `delete[]` was a small memory optimization.

c++ arrays memory-management heap-memory

Share
Improve this question
Follow

edited May 28, 2020 at 2:58
NAND
685 ● 1 ● 10 ● 22

asked Oct 31, 2008 at 3:23

Martin Cote
29.8k ● 15 ● 77 ● 99

6 Answers

Sorted by: Highest score (default)

47

It's so that the destructors of the individual elements will be called. Yes, for arrays of PODs, there isn't much of a difference, but in C++, you can have arrays of objects with non-trivial destructors.

Now, your question is, why not make `new` and `delete` behave like `new[]` and `delete[]` and get rid of `new[]` and `delete[]`? I would go back Stroustrup's "Design and Evolution" book where he said that if you don't use C++ features, you shouldn't

have to pay for them (at run time at least). The way it stands now, a `new` or `delete` will behave as efficiently as `malloc` and `free`. If `delete` had the `delete[]` meaning, there would be some extra overhead at run time (as James Curran pointed out).

Share

Improve this answer

Follow

edited Nov 28, 2014 at 23:42

Deduplicator

45.6k ● 7 ● 71 ● 123

answered Oct 31, 2008 at 3:25

David Nehme

21.6k ● 8 ● 81 ● 121

5 Actualy, when u use new int[1] it stores on the head of the array before it's first data, the size of it. So, using delete instead of delete[] will not free that part of the memory. – Rodrigo Apr 24, 2009 at 13:35

Is there actually modern an implementatnio that couldn't determine (if the standard allowed) the corrrect thing do at runtime regardless of whether delete or delete [] is used? – Pete Fordham May 28, 2020 at 2:30

12

Damn, I missed the whole point of question but I will leave my original answer as a sidenote. Why we have `delete[]` is because long time ago we had `delete[cnt]`, even today if you write `delete[9]` or `delete[cnt]`, the compiler just ignores the thing between `[]` but compiles OK. At that time, C++ was first processed by a front-end and then fed to an ordinary C compiler. They could not do the trick of storing the count somewhere beneath the curtain, maybe they could not even think of it at that time. And for backward compatibility, the compilers most probably used the value given between the `[]` as the count of array, if there is no such value then they got the count from the prefix, so it worked both ways. Later on, we typed nothing between `[]` and everything worked. Today, I do not think `delete[]` is necessary but the implementations demand it that way.

My original answer (that misses the point):

`delete` deletes a single object. `delete[]` deletes an object array. For `delete[]` to work, the implementation keeps the number of elements in the array. I just double-checked this by debugging ASM code. In the implementation (VS2005) I tested, the count was stored as a prefix to the object array.

If you use `delete[]` on a single object, the count variable is garbage so the code crashes. If you use `delete` for an object array, because of some inconsistency, the code crashes. I tested these cases just now !

"`delete` just deletes the memory allocated for the array." statement in another answer is not right. If the object is a class, `delete` will call the DTOR. Just place a breakpoint int the DTOR code and `delete` the object, the breakpoint will hit.

What occurred to me is that, if the compiler & libraries assumed that all the objects allocated by `new` are object arrays, it would be OK to call `delete` for single objects or object arrays. Single objects just would be the special case of an object array having a count of 1. Maybe there is something I am missing, anyway.

Share
Improve this answer
Follow

edited May 28, 2020 at 6:15

NAND
685 ● 1 ● 10 ● 22

answered Oct 31, 2008 at 7:50

Malkocoglu
2,601 ● 2 ● 28 ● 33

9

Since everyone else seems to have missed the point of your question, I'll just add that I had the same thought some year ago, and have never been able to get an answer.

The only thing I can think of is that there's a very tiny bit of extra overhead to treat a single object as an array (an unnecessary "`for(int i=0; i<1; ++i)`")

Share Improve this answer Follow

answered Oct 31, 2008 at 4:51

James Curran
103k ● 37 ● 185 ● 262

3 Plus a tiny bit of memory to store the size. – Loki Astari Oct 31, 2008 at 5:16

1 Yep, I'd bet that the memory overhead was considered unacceptable. Possibly the loop was too. – Steve Jessop Oct 31, 2008 at 15:01

6

Adding this since no other answer currently addresses it:

Array `delete[]` cannot be used on a pointer-to-base class ever -- while the compiler stores the count of objects when you invoke `new[]`, it doesn't store the types or sizes of the objects (as David pointed out, in C++ you rarely pay for a feature you're not using). However, scalar `delete` can safely delete through base class, so it's used both for normal object cleanup and polymorphic cleanup:

```
struct Base { virtual ~Base(); };
struct Derived : Base { };
int main(){
    Base* b = new Derived;
    delete b; // this is good

    Base* b = new Derived[2];
    delete[] b; // bad! undefined behavior
}
```

However, in the opposite case -- non-virtual destructor -- scalar `delete` should be as cheap as possible -- it should not check for number of objects, nor for the type of

object being deleted. This makes delete on a built-in type or plain-old-data type very cheap, as the compiler need only invoke `::operator delete` and nothing else:

```
int main(){
    int * p = new int;
    delete p; // cheap operation, no dynamic dispatch, no conditional branching
}
```

While not an exhaustive treatment of memory allocation, I hope this helps clarify the breadth of memory management options available in C++.

Share Improve this answer Follow

answered Oct 31, 2008 at 20:32

Aaron
3,474 ● 25 ● 26

3

`delete []` ensures that the destructor of each member is called (if applicable to the type) while `delete` just deletes the memory allocated for the array.

Here's a good read: http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=287

~~And no, array sizes are not stored anywhere in C++.~~ (Thanks everyone for pointing out that this statement is inaccurate.)

Share Improve this answer Follow

edited Oct 31, 2008 at 14:45

answered Oct 31, 2008 at 3:27

Ates Goral
140k ● 27 ● 141 ● 191

Don't agree with your last statement. The compiler has to know the array size in order to call the destructor for each object in the array. I think you're confusing this with the fact that C++ doesn't do bounds-checking on arrays. – Mike Spross Oct 31, 2008 at 3:39

Oh true. I thought you were suggesting that the size would be stored as part of the array data structure (buffer). Yes, the compiler would probably have to store the size info somewhere... – Ates Goral Oct 31, 2008 at 4:07

One approach is to store the size and number of elements in the word before the start of the array. This is called a cookie. – Dynite Oct 31, 2008 at 9:16

1 also, delete does call the destructor - for one element. – peterchen Oct 31, 2008 at 10:11

1

I'm a bit confused by Aaron's answer and frankly admit I don't completely understand why and where `delete[]` is needed.

I did some experiments with his sample code (after fixing a few typos). Here are my results. Typos: `~Base` needed a function body `Base *b` was declared twice

```
struct Base { virtual ~Base(){ }>; };
struct Derived : Base { };
int main(){
Base* b = new Derived;
delete b; // this is good

<strike>Base</strike> b = new Derived[2];
delete[] b; // bad! undefined behavior
}
```

Compilation and execution

```
david@Godel:g++ -o atest atest.cpp
david@Godel: ./atest
david@Godel: # No error message
```

Modified program with `delete[]` removed

```
struct Base { virtual ~Base(){}; };
struct Derived : Base { };

int main(){
    Base* b = new Derived;
    delete b; // this is good

    b = new Derived[2];
    delete b; // bad! undefined behavior
}
```

Compilation and execution

```
david@Godel:g++ -o atest atest.cpp
david@Godel: ./atest
atest(30746) malloc: *** error for object 0x1099008c8: pointer being freed was
n
ot allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

Of course, I don't know if `delete[] b` is actually working in the first example; I only know it does not give a compiler error message.

Share

Improve this answer

Follow

edited May 28, 2020 at 7:05

NAND

685 ● 1 ● 10 ● 22

answered Dec 4, 2013 at 14:53

David E.

89 ● 2