C++ Memory management

Asked 16 years, 3 months ago Modified 11 years, 8 months ago

Viewed 7k times

8

I've learned in College that you always have to free your unused Objects but not how you actually do it. For example structuring your code right and so on. Are there any general rules on how to handle pointers in C++?

I'm currently not allowed to use boost. I have to stick to pure c++ because the framework I'm using forbids any use of generics.

c++ memory pointers

Share

Improve this question

Follow

edited Sep 2, 2008 at 1:53

Chris Hanson

55k ● 8 ● 74 ● 104

asked Aug 26, 2008 at 6:50

Alexander Stolz

7,544 ● 13 ● 59 ● 65

I have to stick to pure c++ because the framework I'm using forbids any use of generics. There's certainly a rationale behind this decision? I'd be interested to hear it because off the top of my head I can't think of any reason to forbid

templates completely. – Konrad Rudolph Aug 26, 2008 at 7:57

@KonradRudolph: The official answer to this is: *They are not portable to different operating systems, especially in the way they are supported by compilers and link-editors.* – Alexander Stolz Aug 26, 2008 at 8:05

@AlexanderStolz: It's wrong. I mean, C++ as a whole is not compatible across different OSes (most platforms have a clear and well-defined C ABI, not so C++, especially when you get into details like virtual inheritance representations and so on). Templates don't change anything on that front; instantiated templates are just classes with funny names. Compiler support was a pressing concern a few years ago, but these days there's no excuse not to be using g++ 4.x or VC++ 2005/2008, both of which are more than good enough for any normal usage. – DrPizza Aug 26, 2008 at 18:07

Too bad this is so old. I'd love to hear an explanation about why you can't use (specifically) generics. – jmucchiello Apr 27, 2009 at 1:43

From the documentation of the framework: Do not use Templates. They are not portable to different operating systems, especially in the way they are supported by compilers and link-editors. That's currently all I can say about it – Alexander Stolz Apr 27, 2009 at 11:13

8 Answers

Sorted by: Highest score (default)

14

I have worked with the embedded Symbian OS, which had an excellent system in place for this, based entirely on developer conventions.

1. Only one object will ever own a pointer. By default this is the creator.
2. Ownership can be passed on. To indicate passing of ownership, the object is passed as a pointer in the method signature (e.g. void Foo(Bar *zonk);).
3. The owner will decide when to delete the object.
4. To pass an object to a method just for use, the object is passed as a reference in the method signature (e.g. void Foo(Bat &zonk);).
5. Non-owner classes may store references (never pointers) to objects they are given only when they can be certain that the owner will not destroy it during use.

Basically, if a class simply uses something, it uses a reference. If a class owns something, it uses a pointer.

This worked beautifully and was a pleasure to use. Memory issues were very rare.

Share Improve this answer

Follow

answered Aug 26, 2008 at 7:08

Sander

26.3k ● 3 ● 54 ● 88

I would use std::auto_ptr<> to show that owner ship is being transferred (rather than RAW pointer). Presumably you are already storing in a smart pointer, if sole ownership is expected std::auto_ptr<> is perfect. – Loki Astari Oct 2, 2008 at 15:22

The OP said he couldn't use generics, which presumably rules out any smart pointer type. – Sam Stokes Nov 25, 2008

at 19:10

Wont application core dump if the object pointed by the reference is deleted by mistake? If it had been a pointer, we could check for null.(provided the deleter sets it to null.)
– balki Oct 10, 2011 at 8:45

@balki It might; it might not; it might make toasters rain from the sky: it's UB. But Sander said references are only used if there's a guarantee they don't become dangling, so that's a code quality issue. And more important, pointers are no better: deletion of the object by another entity doesn't set the observer's copy of the pointer to `nullptr`, so it can't be tested for validity anymore than a reference can (read: `nullptr`ness is a property of an address, not the data at that address) ...unless each observer ptr is registered by ref w/ the owner for later updating or something convoluted
– underscore_d Jan 17, 2017 at 1:24

5

Rules:

1. Wherever possible, use a smart pointer. Boost has some good ones.
2. If you can't use a smart pointer, null out your pointer after deleting it.
3. Never work anywhere that won't let you use rule 1.

If someone disallows rule 1, remember that if you grab someone else's code, change the variable names and delete the copyright notices, no-one will ever notice. Unless it's a school project, where they actually check for that kind of shenanigans with quite sophisticated tools. See also, this question.

Share Improve this answer

Follow

edited May 23, 2017 at 12:33

Community Bot

1 ● 1

answered Aug 26, 2008 at 6:52

Josh

8,016 ● 5 ● 43 ● 63

-1 for the nulling. I won't discuss it here, but it is **not** good practice. If the code is written fairly well, it should not be necessary, as you shouldn't delete before it is sure that no **other** references/pointers to your allocated memory block exist anymore. – Rudy Velthuis Jul 31, 2016 at 2:13

3

I would add another rule here:

- Don't new/delete an object when an automatic object will do just fine.

We have found that programmers who are new to C++, or programmers coming over from languages like Java, seem to learn about new and then obsessively use it whenever they want to create any object, regardless of the context. This is especially pernicious when an object is created locally within a function purely to do something useful. Using new in this way can be detrimental to performance and can make it all too easy to introduce silly memory leaks when the corresponding delete is forgotten. Yes, smart pointers can help with the latter but it won't solve the performance issues (assuming that new/delete or an equivalent is used behind the scenes).

Interestingly (well, maybe), we have found that delete often tends to be more expensive than new when using Visual C++.

Some of this confusion also comes from the fact that functions they call might take pointers, or even smart pointers, as arguments (when references would perhaps be better/clearer). This makes them think that they need to "create" a pointer (a lot of people seem to think that this is what new does) to be able to pass a pointer to a function. Clearly, this requires some rules about how APIs are written to make calling conventions as unambiguous as possible, which are reinforced with clear comments supplied with the function prototype.

Share Improve this answer

Follow

answered Aug 26, 2008 at 17:09

Nick

761 ● 4 ● 9

Picking up from Nice "This makes them think that they need to "create" a pointer (a lot of people seem to think that this is what new does) to be able to pass a pointer to a function.)" Maybe, but I think its because most universities teach Java first. In Java you have to "new" everything and it's hard to break the habit, especially when the two languages look so similar, but are in fact completely different. For real fun, try working with both at the same time! – Chris Huang-Leaver Oct 2, 2008 at 14:33

In the general case (resource management, where resource is not necessarily memory), you need to be

2

familiar with the RAII pattern. This is one of the most important pieces of information for C++ developers.

Share Improve this answer

Follow

answered Aug 26, 2008 at 7:37

On Freund

4,436 ● 2 ● 24 ● 31

2

In general, avoid allocating from the heap unless you have to. If you have to, use reference counting for objects that are long-lived and need to be shared between diverse parts of your code.

Sometimes you need to allocate objects dynamically, but they will only be used within a certain span of time. For example, in a previous project I needed to create a complex in-memory representation of a database schema -- basically a complex cyclic graph of objects. However, the graph was only needed for the duration of a database connection, after which all the nodes could be freed in one shot. In this kind of scenario, a good pattern to use is something I call the "local GC idiom." I'm not sure if it has an "official" name, as it's something I've only seen in my own code, and in Cocoa (see NSAutoreleasePool in Apple's Cocoa reference).

In a nutshell, you create a "collector" object that keeps pointers to the temporary objects that you allocate using new. It is usually tied to some scope in your program, either a static scope (e.g. -- as a stack-allocated object that implements the RAII idiom) or a dynamic one (e.g. --

tied to the lifetime of a database connection, as in my previous project). When the "collector" object is freed, its destructor frees all of the objects that it points to.

Also, like DrPizza I think the restriction to not use templates is too harsh. However, having done a lot of development on ancient versions of Solaris, AIX, and HP-UX (just recently - yes, these platforms are still alive in the Fortune 50), I can tell you that if you really care about portability, you should use templates as little as possible. Using them for containers and smart pointers ought to be ok, though (it worked for me). Without templates the technique I described is more painful to implement. It would require that all objects managed by the "collector" derive from a common base class.

Share Improve this answer

Follow

answered Aug 26, 2008 at 19:18

Bruce Johnston

8,624 ● 3 ● 34 ● 43

While a local GC idiom sounds nice, it is not necessary. Manual memory management is not so hard to do well.
– Rudy Velthuis Jul 31, 2016 at 2:15

1

G'day,

I'd suggest reading the relevant sections of "Effective C++" by Scott Meyers. Easy to read and he covers some interesting gotchas to trap the unwary.

I'm also intrigued by the lack of templates. So no STL or Boost. Wow.

BTW Getting people to agree on conventions is an excellent idea. As is getting everyone to agree on conventions for OOD. BTW The latest edition of Effective C++ doesn't have the excellent chapter about OOD conventions that the first edition had which is a pity, e.g. conventions such as public virtual inheritance *always* models an "isa" relationship.

Rob

Share Improve this answer

Follow

answered Aug 26, 2008 at 9:39

Rob Wells

37k ● 13 ● 84 ● 147

0

- When you have to use manage memory manually, make sure you call delete in the same scope/function/class/module, which ever applies first, e.g.:
- Let the caller of a function allocate the memory that is filled by it, do not return new'ed pointers.
- Always call delete in the same exe/dll as you called new in, because otherwise you may have problems with heap corruptions (different incompatible runtime libraries).

Share Improve this answer

Follow

answered Aug 26, 2008 at 7:06

Timbo

28k • 11 • 51 • 75

0

you could derive everything from some base class that implement smart pointer like functionality (using ref()/unref() methods and a counter.

All points highlighted by @Timbo are important when designing that base class.

Share Improve this answer

Follow

answered Aug 26, 2008 at 8:19

mkrus

825 • 1 • 8 • 16