# Allocation

**The problem**

C and C++ memory allocation and freeing was a manual process;

[unix beard]

Any possible resemblance to a real person is completely coincidental.

malloc/free or new/delete

[stack of boxes]

Stack allocation is much easier, memory is freed and destructors are called automatically. We will come back to this.

But what if your object is too big for the stack? What if you need to share it with others and it has an indefinite lifetime?

[warehouse]

Easy in garbage collected languages where the GC is the owner of all objects, e.g. C# but C++ is not garbage collected and has no mechanism to support full GC.

‘Conservative GC’ does exist but is not completely effective. I’ll leave you to draw your own conclusions there.

[kitten]

The idea of the ownership of a pointer when the GC is not there to own it for you.

Pointers, like kittens, can be unruly and have a habit of wandering off.

Also, a side note, arrays are just pointers and the same applies. Avoid. std::vector is better in every regard. And std:vector plays nicely with the clever pointers I will describe later.

**Object too big for stack**

[Robot hand]

auto\_ptr<T> is a stack allocated reference to heap allocated data which overload the pointer operators so that they (mostly) behave as pointers. If the auto\_ptr is still pointing at something when it goes out of scope then the heap object will also be freed/deleted.

auto\_ptr does not copy in the usual sense. The underlying pointer is transferred, not copied to ensure there is a single owner. When you go on holiday you leave your kitten with a friend, you don’t duplicate it. There can only be one owner even if only temporarily. This is annoying but removes a number of bugs or at least, makes them more obvious. auto\_ptr was a step in the right direction but we can do better.

**A better auto\_ptr**

**Demo the code.**

shared\_ptr is an auto\_ptr which can be copied. The simultaneous ownership of a single resource (a pointer) by multiple shared\_ptr instances is handled by reference counting.

However, this is not COM (scream!)

You will ordinarily be unaware of the reference counting which is handled by the shared\_ptr instances themselves and a counter attached to the actual resource in the heap.

Reference counting does not play well with circular structures which will validate their own existence forever.

For this reason, and to permit passing references which do not determine object lifetimes, there is weak\_ptr. This does not increment the reference count except when the weak\_ptr is ‘locked’ which temporarily creates from it a shared\_ptr and a fully-fledged reference. Lock may fail and return nullptr if the referred object has expired.

(Actually it returns a wrapped null pointer if you want to be precise)

You can still call legacy code using get() to obtain a real pointer. As long as the code respects the guaranteed object lifetime which ends when that code returns, all will be well.

Also note that the reference count is only maintained when you copy the pointer. Passing by reference does not modify the reference count. So that is simple and obvious! Passing these smart pointers by reference should be considered a bug as it is confusing and pointless.

Using shared\_ptr still requires careful thought and pointers will always be complicated but they do make things easier to manage than a bare address. You should still look for simpler alternatives if they exist. Which brings us to…

**The best way**

Best of all just create items on the stack and pass them by value or as a reference. This avoids any explicit use of pointers and allows the compiler to control any pointer used by the implementation to execute the code. Your code will always use implicit pointers but the compiler is much, much cleverer at the book keeping required to use them effectively. You also completely avoid the dreaded null reference exception.

So what is the better approach than a shared pointer?

Take a piece of data and **copy** it to interested agents. They can read and update it at their leisure and destroy it when they are done. They can make concurrent updates with no ill effects. They will not see the others changes and this is not a bad thing. If they need to share changes then you need a proper update mechanism not just a free-for-all on a shared datum. (Even if you *can* make this work the next edit to that clever code *will* break it.) Also note that the overhead in setting up locking, semaphores etc. can be substantial and less deterministic than a simple block memory copy.

There is a reason that single assignment to immutable data structures is “the next big thing”

Don’t pre-judge the compiler or libraries. Copying structures is seemingly less efficient that passing pointers but bear in mind that the compiler and library writers are aware of these patterns of usage and may implement copy on write strategies to avoid unnecessary copying and memory use. As a general rule, write simple code and optimise only if necessary.