Excerpt from Effective Modern C++ by Scott Meyers

Chapter 4 Introduction

Raw pointers are powerful tools, to be sure, but decades of experience have demonstrated that with only the slightest lapse in concentration or discipline, these tools can turn on their ostensible masters.

Smart pointers are one way to address these issues. Smart pointers are wrappers around raw pointers that act much like the raw pointers they wrap, but that avoid many of their pitfalls. You should therefore prefer smart pointers to raw pointers. Smart pointers can do virtually everything raw pointers can, but with far fewer opportunities for error.

There are four smart pointers in C++11: std::auto_ptr, std::unique_ptr, std::shared_ptr, and std::weak_ptr. All are designed to help manage the lifetimes of dynamically allocated objects, i.e., to avoid resource leaks by ensuring that such objects are destroyed in the appropriate manner at the appropriate time (including in the event of exceptions).

std::auto_ptr is a deprecated leftover from C++98. It was an attempt to standardize what later became C++11's std::unique_ptr. Doing the job right required move semantics, but C++98 didn't have them. As a workaround, std::auto_ptr co-opted its copy operations for moves. This led to surprising code (copying a std::auto_ptrsets it to null!) and frustrating usage restrictions (e.g., it's not possible to store std::auto_ptrs in containers).

std::unique_ptr does everything std::auto_ptr does, plus more. It does it as efficiently, and it does it without warping what it means to copy an object. It's better than std::auto_ptr in every way. The only legitimate use case for std::auto_ptr is a need to compile code with C++98 compilers. Unless you have that constraint, you should replace std::auto_ptr with std::unique_ptr and never look back.

The smart pointer APIs are remarkably varied. About the only functionality common to all is default construction. Because comprehensive references for these APIs are widely available, I'll focus my discussions on information that's often missing from API overviews, e.g., noteworthy use cases, runtime cost analyses, etc. Mastering such information can be the difference between merely using these smart pointers and using them effectively.

Item 18: Use std::unique_ptr for exclusive-ownership resource management.

When you reach for a smart pointer, std::unique_ptr should generally be the one closest at hand. It's reasonable to assume that, by default, std::unique_ptrs are the same size as raw pointers, and for most operations (including dereferencing), they execute exactly the same instructions. This means you can use them even in situations where memory and cycles are tight. If a raw pointer is small enough and fast enough for you, a std::unique_ptr almost certainly is, too.

std::unique_ptr embodies exclusive ownership semantics. A non-null std::unique_ptr always owns what it points to. Moving a std::unique_ptr transfers ownership from the source pointer to the destination pointer. (The source pointer is set to null.) Copying a std::unique_ptr isn't allowed, because if you could copy a std::unique_ptr, you'd end up with two std::unique_ptrs to the same resource, each thinking it owned (and should therefore destroy) that resource. std::unique_ptr is thus a move-only type. Upon destruction, a non-null std::unique_ptr destroys its resource. By default, resource destruction is accomplished by applying delete to the raw pointer inside the std::unique_ptr.