CHAPTER 4

Smart Pointers

Poets and songwriters have a thing about love. And sometimes about counting. Occasionally both. Inspired by the rather different takes on love and counting by Elizabeth Barrett Browning ("How do I love thee? Let me count the ways.") and Paul Simon ("There must be 50 ways to leave your lover."), we might try to enumerate the reasons why a raw pointer is hard to love:

1. Its declaration doesn't indicate whether it points to a single object or to an array.
2. Its declaration reveals nothing about whether you should destroy what it points to when you're done using it, i.e., if the pointer *owns* the thing it points to.
3. If you determine that you should destroy what the pointer points to, there's no way to tell how. Should you use `delete`, or is there a different destruction mechanism (e.g., a dedicated destruction function the pointer should be passed to)?
4. If you manage to find out that `delete` is the way to go, Reason 1 means it may not be possible to know whether to use the single-object form ("`delete`") or the array form ("`delete []`"). If you use the wrong form, results are undefined.
5. Assuming you ascertain that the pointer owns what it points to and you discover how to destroy it, it's difficult to ensure that you perform the destruction *exactly once* along every path in your code (including those due to exceptions). Missing a path leads to resource leaks, and doing the destruction more than once leads to undefined behavior.
6. There's typically no way to tell if the pointer dangles, i.e., points to memory that no longer holds the object the pointer is supposed to point to. Dangling pointers arise when objects are destroyed while pointers still point to them.

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_ptr` sets it to null!) and frustrating usage restrictions (e.g., it's not possible to store `std::auto_ptr`s 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_ptr`s 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_ptr`s 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`.

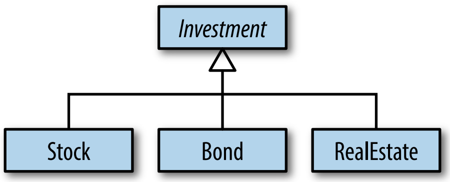
A common use for `std::unique_ptr` is as a factory function return type for objects in a hierarchy. Suppose we have a hierarchy for types of investments (e.g., stocks, bonds, real estate, etc.) with a base class `Investment`.

```
class Investment { … };

class Stock:
  public Investment { … };

class Bond:
  public Investment { … };

class RealEstate:
  public Investment { … };
```



A factory function for such a hierarchy typically allocates an object on the heap and returns a pointer to it, with the caller being responsible for deleting the object when it's no longer needed. That's a perfect match for `std::unique_ptr`, because the caller acquires responsibility for the resource returned by the factory (i.e., exclusive ownership of it), and the `std::unique_ptr` automatically deletes what it points to when the `std::unique_ptr` is destroyed. A factory function for the `Investment` hierarchy could be declared like this:

```
template<typename... Ts>          // return std::unique_ptr
std::unique_ptr<Investment>       // to an object created
makeInvestment(Ts&&... params);   // from the given args
```

Callers could use the returned `std::unique_ptr` in a single scope as follows,

```
{
  …
```

```
    auto pInvestment =                 // pInvestment is of type
      makeInvestment( arguments );     // std::unique_ptr<Investment>

    …

  }                                    // destroy *pInvestment
```

but they could also use it in ownership-migration scenarios, such as when the `std::unique_ptr` returned from the factory is moved into a container, the container element is subsequently moved into a data member of an object, and that object is later destroyed. When that happens, the object's `std::unique_ptr` data member would also be destroyed, and its destruction would cause the resource returned from the factory to be destroyed. If the ownership chain got interrupted due to an exception or other atypical control flow (e.g., early function return or `break` from a loop), the `std::unique_ptr` owning the managed resource would eventually have its destructor called,[1] and the resource it was managing would thereby be destroyed.

By default, that destruction would take place via `delete`, but, during construction, `std::unique_ptr` objects can be configured to use *custom deleters*: arbitrary functions (or function objects, including those arising from lambda expressions) to be invoked when it's time for their resources to be destroyed. If the object created by `makeInvestment` shouldn't be directly `delete`d, but instead should first have a log entry written, `makeInvestment` could be implemented as follows. (An explanation follows the code, so don't worry if you see something whose motivation is less than obvious.)

```
  auto delInvmt = [](Investment* pInvestment)       // custom
                  {                                 // deleter
                    makeLogEntry(pInvestment);      // (a lambda
                    delete pInvestment;             // expression)
                  };

  template<typename... Ts>                          // revised
  std::unique_ptr<Investment, decltype(delInvmt)>   // return type
  makeInvestment(Ts&&... params)
  {
    std::unique_ptr<Investment, decltype(delInvmt)> // ptr to be
      pInv(nullptr, delInvmt);                      // returned
```

1 There are a few exceptions to this rule. Most stem from abnormal program termination. If an exception propagates out of a thread's primary function (e.g., `main`, for the program's initial thread) or if a `noexcept` specification is violated (see Item 14), local objects may not be destroyed, and if `std::abort` or an exit function (i.e., `std::_Exit`, `std::exit`, or `std::quick_exit`) is called, they definitely won't be.

```
  if ( /* a Stock object should be created */ )
  {
    pInv.reset(new Stock(std::forward<Ts>(params)...));
  }
  else if ( /* a Bond object should be created */ )
  {
    pInv.reset(new Bond(std::forward<Ts>(params)...));
  }
  else if ( /* a RealEstate object should be created */ )
  {
    pInv.reset(new RealEstate(std::forward<Ts>(params)...));
  }

  return pInv;
}
```

In a moment, I'll explain how this works, but first consider how things look if you're a caller. Assuming you store the result of the `makeInvestment` call in an `auto` variable, you frolic in blissful ignorance of the fact that the resource you're using requires special treatment during deletion. In fact, you veritably bathe in bliss, because the use of `std::unique_ptr` means you need not concern yourself with when the resource should be destroyed, much less ensure that the destruction happens exactly once along every path through the program. `std::unique_ptr` takes care of all those things automatically. From a client's perspective, `makeInvestment`'s interface is sweet.

The implementation is pretty nice, too, once you understand the following:

- `delInvmt` is the custom deleter for the object returned from `makeInvestment`. All custom deletion functions accept a raw pointer to the object to be destroyed, then do what is necessary to destroy that object. In this case, the action is to call `makeLogEntry` and then apply `delete`. Using a lambda expression to create `delInvmt` is convenient, but, as we'll see shortly, it's also more efficient than writing a conventional function.
- When a custom deleter is to be used, its type must be specified as the second type argument to `std::unique_ptr`. In this case, that's the type of `delInvmt`, and that's why the return type of `makeInvestment` is `std::unique_ptr<Investment, decltype(delInvmt)>`. (For information about `decltype`, see Item 3.)
- The basic strategy of `makeInvestment` is to create a null `std::unique_ptr`, make it point to an object of the appropriate type, and then return it. To associate the custom deleter `delInvmt` with `pInv`, we pass that as its second constructor argument.

- Attempting to assign a raw pointer (e.g., from new) to a std::unique_ptr won't compile, because it would constitute an implicit conversion from a raw to a smart pointer. Such implicit conversions can be problematic, so C++11's smart pointers prohibit them. That's why reset is used to have pInv assume ownership of the object created via new.
- With each use of new, we use std::forward to perfect-forward the arguments passed to makeInvestment (see Item 25). This makes all the information provided by callers available to the constructors of the objects being created.
- The custom deleter takes a parameter of type Investment*. Regardless of the actual type of object created inside makeInvestment (i.e., Stock, Bond, or Real Estate), it will ultimately be deleted inside the lambda expression as an Investment* object. This means we'll be deleting a derived class object via a base class pointer. For that to work, the base class—Investment—must have a virtual destructor:

  ```
  class Investment {
  public:
    …                                             // essential
    virtual ~Investment();                        // design
    …                                             // component!
  };
  ```

In C++14, the existence of function return type deduction (see Item 3) means that makeInvestment could be implemented in this simpler and more encapsulated fashion:

```
template<typename... Ts>
auto makeInvestment(Ts&&... params)              // C++14
{
  auto delInvmt = [](Investment* pInvestment)    // this is now
                  {                              // inside
                    makeLogEntry(pInvestment);   // make-
                    delete pInvestment;          // Investment
                  };

  std::unique_ptr<Investment, decltype(delInvmt)>  // as
    pInv(nullptr, delInvmt);                       // before

  if ( … )                                         // as before
  {
    pInv.reset(new Stock(std::forward<Ts>(params)...));
  }
  else if ( … )                                    // as before
```

```
    {
      pInv.reset(new Bond(std::forward<Ts>(params)...));
    }
    else if ( … )                                   // as before
    {
      pInv.reset(new RealEstate(std::forward<Ts>(params)...));
    }
    return pInv;                                    // as before
  }
```

I remarked earlier that, when using the default deleter (i.e., `delete`), you can reasonably assume that `std::unique_ptr` objects are the same size as raw pointers. When custom deleters enter the picture, this may no longer be the case. Deleters that are function pointers generally cause the size of a `std::unique_ptr` to grow from one word to two. For deleters that are function objects, the change in size depends on how much state is stored in the function object. Stateless function objects (e.g., from lambda expressions with no captures) incur no size penalty, and this means that when a custom deleter can be implemented as either a function or a captureless lambda expression, the lambda is preferable:

```
auto delInvmt1 = [](Investment* pInvestment)      // custom
                 {                                // deleter
                   makeLogEntry(pInvestment);     // as
                   delete pInvestment;            // stateless
                 };                               // lambda

template<typename... Ts>                          // return type
std::unique_ptr<Investment, decltype(delInvmt1)>  // has size of
makeInvestment(Ts&&... args);                     // Investment*

void delInvmt2(Investment* pInvestment)           // custom
{                                                 // deleter
  makeLogEntry(pInvestment);                      // as function
  delete pInvestment;
}

template<typename... Ts>                // return type has
std::unique_ptr<Investment,             // size of Investment*
                void (*)(Investment*)>  // plus at least size
makeInvestment(Ts&&... params);         // of function pointer!
```

Function object deleters with extensive state can yield `std::unique_ptr` objects of significant size. If you find that a custom deleter makes your `std::unique_ptr`s unacceptably large, you probably need to change your design.

Factory functions are not the only common use case for `std::unique_ptr`s. They're even more popular as a mechanism for implementing the Pimpl Idiom. The code for that isn't complicated, but in some cases it's less than straightforward, so I'll refer you to Item 22, which is dedicated to the topic.

`std::unique_ptr` comes in two forms, one for individual objects (`std::unique_ptr<T>`) and one for arrays (`std::unique_ptr<T[]>`). As a result, there's never any ambiguity about what kind of entity a `std::unique_ptr` points to. The `std::unique_ptr` API is designed to match the form you're using. For example, there's no indexing operator (`operator[]`) for the single-object form, while the array form lacks dereferencing operators (`operator*` and `operator->`).

The existence of `std::unique_ptr` for arrays should be of only intellectual interest to you, because `std::array`, `std::vector`, and `std::string` are virtually always better data structure choices than raw arrays. About the only situation I can conceive of when a `std::unique_ptr<T[]>` would make sense would be when you're using a C-like API that returns a raw pointer to a heap array that you assume ownership of.

`std::unique_ptr` is the C++11 way to express exclusive ownership, but one of its most attractive features is that it easily and efficiently converts to a `std::shared_ptr`:

```
std::shared_ptr<Investment> sp =    // converts std::unique_ptr
  makeInvestment( arguments );      // to std::shared_ptr
```

This is a key part of why `std::unique_ptr` is so well suited as a factory function return type. Factory functions can't know whether callers will want to use exclusive-ownership semantics for the object they return or whether shared ownership (i.e., `std::shared_ptr`) would be more appropriate. By returning a `std::unique_ptr`, factories provide callers with the most efficient smart pointer, but they don't hinder callers from replacing it with its more flexible sibling. (For information about `std::shared_ptr`, proceed to Item 19.)

Things to Remember

- `std::unique_ptr` is a small, fast, move-only smart pointer for managing resources with exclusive-ownership semantics.
- By default, resource destruction takes place via `delete`, but custom deleters can be specified. Stateful deleters and function pointers as deleters increase the size of `std::unique_ptr` objects.
- Converting a `std::unique_ptr` to a `std::shared_ptr` is easy.