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 wasn't 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*.

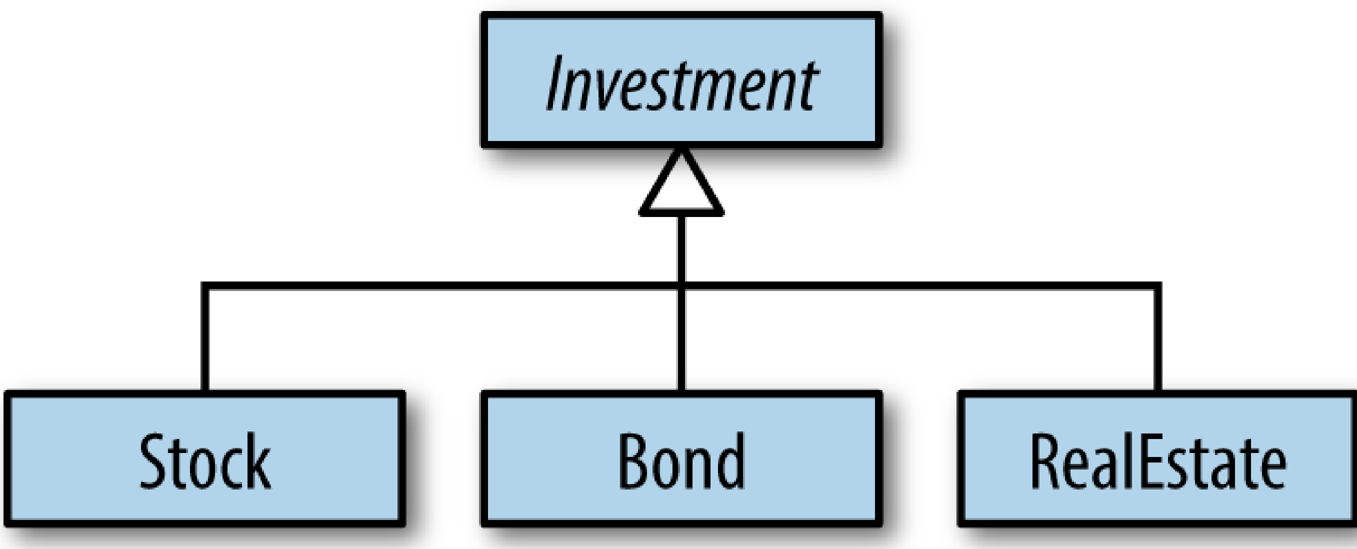
Item 18: Use `std::unique_ptr` for exclusive-ownership resource management.

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` transfersownership 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

  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 `RealEstate`), it will ultimately be `delete`d 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) typically incur no size penalty when used as deleters, 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.

Item 19: Use `std::shared_ptr` for shared-ownership resource management.

Programmers using languages with garbage collection point and laugh at what C++ programmers go through to prevent resource leaks. "How primitive!" they jeer. "Didn't you get the memo from Lisp in the 1960s? Machines should manage resource lifetimes, not humans." C++ developers roll their eyes. "You mean the memo where the only resource is memory and the timing of resource reclamation is nondeterministic? We prefer the generality and predictability of destructors, thank you." But our bravado is part bluster. Garbage collection really is convenient, and manual lifetime management really can seem akin to constructing a mnemonic memory circuit using stone knives and bear skins. Why can't we have the best of both worlds: a system that works automatically (like garbage collection), yet applies to all resources and has predictable timing (like destructors)?

`std::shared_ptr` is the C++11 way of binding these worlds together. An object accessed via `std::shared_ptr`s has its lifetime managed by those pointers through *shared ownership*. No specific `std::shared_ptr` owns the object. Instead, all `std::shared_ptr`s pointing to it collaborate to ensure its destruction at the point where it's no longer needed. When the last `std::shared_ptr` pointing to an object stops pointing there (e.g., because the `std::shared_ptr` is destroyed or made to point to a different object), that `std::shared_ptr` destroys the object it points to. As with garbage collection, clients need not concern themselves with managing the lifetime of pointed-to objects, but as with destructors, the timing of the objects' destruction is deterministic.

A `std::shared_ptr` can tell whether it's the last one pointing to a resource by consulting the resource's *reference count*, a value associated with the resource that keeps track of how many `std::shared_ptr`s point to it. `std::shared_ptr` constructors increment this count (usually—see below), `std::shared_ptr` destructors decrement it, and copy assignment operators do both.

(If sp1 and sp2 are std::shared_ptrs to different objects, the assignment "sp1 = sp2;" modifies sp1 such that it points to the object pointed to by sp2. The net effect of the assignment is that the reference count for the object originally pointed to by sp1 is decremented, while that for the object pointed to by sp2 is incremented.) If a std::shared_ptr sees a reference count of zero after performing a decrement, no more std::shared_ptrs point to the resource, so the std::shared_ptrdestroys it.

The existence of the reference count has performance implications:

- **std::shared_ptrs are twice the size of a raw pointer**, because they internally contain a raw pointer to the resource as well as a raw pointer to the resource's reference count.[2]
- **Memory for the reference count must be dynamically allocated**. Conceptually, the reference count is associated with the object being pointed to, but pointed-to objects know nothing about this. They thus have no place to store a reference count. (A pleasant implication is that any object—even those of built-in types—may be managed by std::shared_ptrs.) Item 21 explains that the cost of the dynamic allocation is avoided when the std::shared_ptr is created by std::make_shared, but there are situations where std::make_shared can't be used. Either way, the reference count is stored as dynamically allocated data.
- **Increments and decrements of the reference count must be atomic**, because there can be simultaneous readers and writers in different threads. For example, a std::shared_ptr pointing to a resource in one thread could be executing its destructor (hence decrementing the reference count for the resource it points to), while, in a different thread, a std::shared_ptr to the same object could be copied (and therefore incrementing the same reference count). Atomic operations are typically slower than non-atomic operations, so even though reference counts are usually only a word in size, you should assume that reading and writing them is comparatively costly.

Did I pique your curiosity when I wrote that std::shared_ptr constructors only "usually" increment the reference count for the object they point to? Creating a std::shared_ptrpointing to an object always yields one more std::shared_ptr pointing to that object, so why mustn't we *always* increment the reference count?

Move construction, that's why. Move-constructing a std::shared_ptr from another std::shared_ptr sets the source std::shared_ptr to null, and that means that the old std::shared_ptr stops pointing to the resource at the moment the new std::shared_ptr starts. As a result, no reference count manipulation is required. Moving std::shared_ptrs is therefore faster than copying them: copying requires incrementing the reference count, but moving doesn't. This is as true for assignment as for construction, so move construction is faster than copy construction, and move assignment is faster than copy assignment.

Like `std::unique_ptr` (see Item 18), `std::shared_ptr` uses `delete` as its default resource-destruction mechanism, but it also supports custom deleters. The design of this support differs from that for `std::unique_ptr`, however. For `std::unique_ptr`, the type of the deleter is part of the type of the smart pointer. For `std::shared_ptr`, it's not:

```
auto loggingDel = [](Widget *pw)        // custom deleter
                  {                     // (as in Item 18)
                    makeLogEntry(pw);
                    delete pw;
                  };

std::unique_ptr<                        // deleter type is
  Widget, decltype(loggingDel)          // part of ptr type
  > upw(new Widget, loggingDel);

std::shared_ptr<Widget>                 // deleter type is not
  spw(new Widget, loggingDel);          // part of ptr type
```

The `std::shared_ptr` design is more flexible. Consider two `std::shared_ptr<Widget>`s, each with a custom deleter of a different type (e.g., becausethe custom deleters are specified via lambda expressions):

```
auto customDeleter1 = [](Widget *pw) { … };     // custom deleters,
auto customDeleter2 = [](Widget *pw) { … };     // each with a
                                                // different type

std::shared_ptr<Widget> pw1(new Widget, customDeleter1);
std::shared_ptr<Widget> pw2(new Widget, customDeleter2);
```

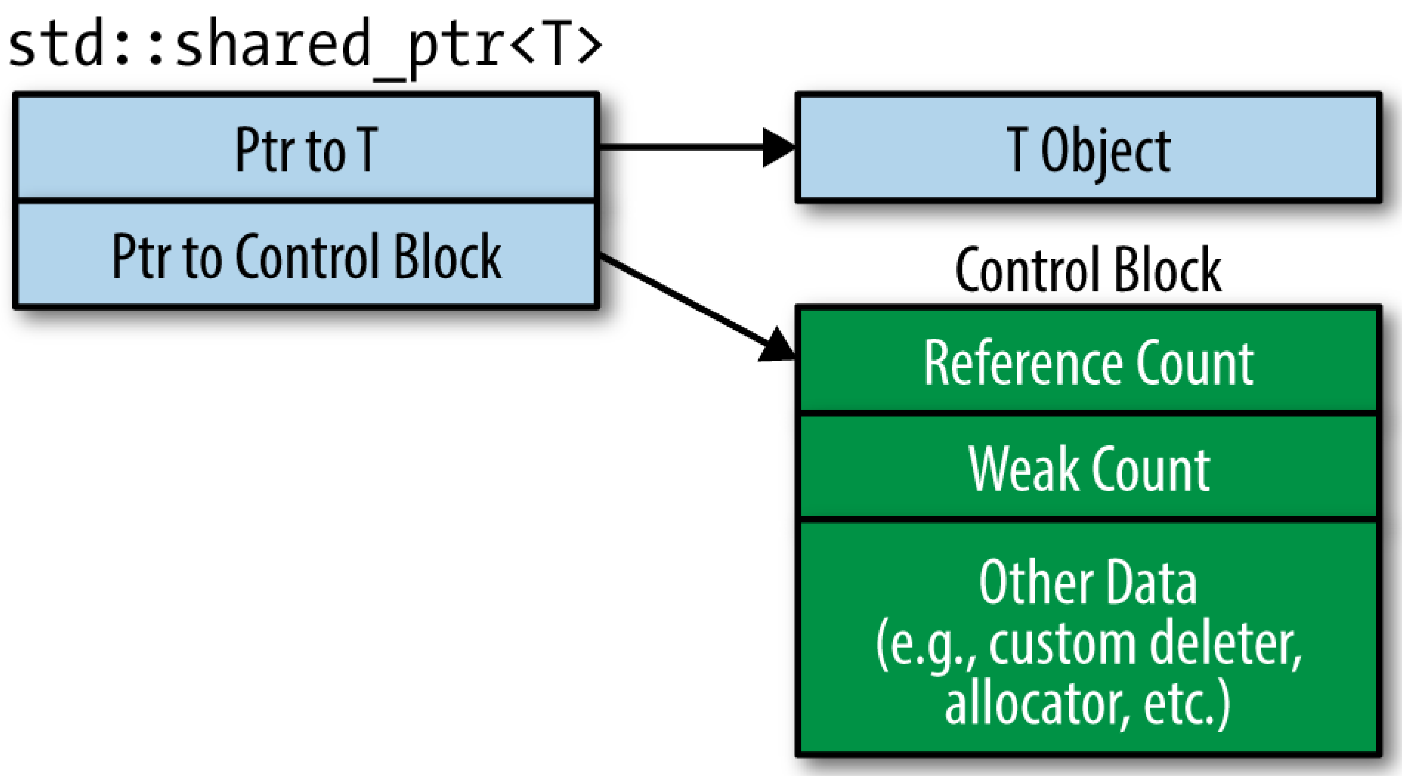
Because `pw1` and `pw2` have the same type, they can be placed in a container of objects of that type:

```
std::vector<std::shared_ptr<Widget>> vpw{ pw1, pw2 };
```

They could also be assigned to one another, and they could each be passed to a function taking a parameter of type `std::shared_ptr<Widget>`. None of these things can be done with `std::unique_ptr`s that differ in the types of their custom deleters, because the type of the custom deleter would affect the type of the `std::unique_ptr`.

In another difference from `std::unique_ptr`, specifying a custom deleter doesn't change the size of a `std::shared_ptr` object. Regardless of deleter, a `std::shared_ptr` object is two pointers in size. That's great news, but it should make you vaguely uneasy. Custom deleters can be function objects, and function objects can contain arbitrary amounts of data. That means they can be arbitrarily large. How can a `std::shared_ptr` refer to a deleter of arbitrary size without using any more memory?

It can't. It may have to use more memory. However, that memory isn't part of the `std::shared_ptr` object. It's on the heap or, if the creator of the `std::shared_ptr` took advantage of `std::shared_ptr` support for custom allocators, it's wherever the memory managed by the allocator is located. I remarked earlier that a `std::shared_ptr` object contains a pointer to the reference count for the object it points to. That's true, but it's a bit misleading, because the reference count is part of a larger data structure known as the *control block*. There's a control block for each object managed by `std::shared_ptrs`. The control block contains, in addition to the reference count, a copy of the custom deleter, if one has been specified. If a custom allocator was specified, the control block contains a copy of that, too. The control block contains additional data, including, as Item 21explains, a secondary reference count known as the weak count, but we'll largely ignore such data in this Item. We can envision the memory associated with a `std::shared_ptr<T>` object as looking like this:



An object's control block is set up by the function creating the first `std::shared_ptr` to the object. At least that's what's supposed to happen. In general, it's impossible for a function creating a `std::shared_ptr` to an object to know whether some other `std::shared_ptr` already points to that object, so the following rules for control block creation are used:

- **`std::make_shared` (see Item 21) always creates a control block.** It manufactures a new object to point to, so there is certainly no control block for that object at the time `std::make_shared` is called.
- **A control block is created when a `std::shared_ptr` is constructed from a unique-ownership pointer (i.e., a `std::unique_ptr` or `std::auto_ptr`).**Unique-ownership pointers don't use control blocks, so there should be no control block for the pointed-to object. (As part of its construction, the `std::shared_ptr`assumes ownership of the pointed-to object, so the unique-ownership pointer is set to null.)
- **When a `std::shared_ptr` constructor is called with a raw pointer, it creates a control block.** If you wanted to create a `std::shared_ptr` from an object that already had a control block, you'd presumably pass a `std::shared_ptr` or a `std::weak_ptr` (see Item 20) as a constructor argument, not a raw pointer. `std::shared_ptr` constructors taking `std::shared_ptrs` or `std::weak_ptrs` as constructor arguments don't create new control blocks, because they can rely on the smart pointers passed to them to point to any necessary control blocks.

A consequence of these rules is that constructing more than one `std::shared_ptr` from a single raw pointer gives you a complimentary ride on the particle accelerator of undefined behavior, because the pointed-to object will have multiple control blocks. Multiple control blocks means multiple reference counts, and multiple reference counts means the object will be destroyed multiple times (once for each reference count). That means that code like this is bad, bad, bad:

```
auto pw = new Widget;                          // pw is raw ptr

…

std::shared_ptr<Widget> spw1(pw, loggingDel);  // create control
                                               // block for *pw
…

std::shared_ptr<Widget> spw2(pw, loggingDel);  // create 2nd
                                               // control block
                                               // for *pw!
```

The creation of the raw pointer pw to a dynamically allocated object is bad, because it runs contrary to the advice behind this entire chapter: to prefer smart pointers to raw pointers. (If you've forgotten the motivation for that advice, refresh your memory here.) But set that aside. The line creating pw is a stylistic abomination, but at least it doesn't cause undefined program behavior.

Now, the constructor for spw1 is called with a raw pointer, so it creates a control block (and thereby a reference count) for what's pointed to. In this case, that's *pw (i.e., the object pointed to by pw). In and of itself, that's okay, but the constructor for spw2 is called with the same raw pointer, so it also creates a control block (hence a reference count) for *pw. *pw thus has two reference counts, each of which will eventually become zero, and that will ultimately lead to an attempt to destroy *pw twice. The second destruction is responsible for the undefined behavior.

There are at least two lessons regarding std::shared_ptr use here. First, try to avoid passing raw pointers to a std::shared_ptr constructor. The usual alternative is to use std::make_shared (see Item 21), but in the example above, we're using custom deleters, and that's not possible with std::make_shared. Second, if you must pass a raw pointer to a std::shared_ptr constructor, pass the result of new directly instead of going through a raw pointer variable. If the first part of the code above were rewritten like this,

```
std::shared_ptr<Widget> spw1(new Widget,     // direct use of new
                             loggingDel);
```

it'd be a lot less tempting to create a second std::shared_ptr from the same raw pointer. Instead, the author of the code creating spw2 would naturally use spw1 as an initialization argument (i.e., would call the std::shared_ptr copy constructor), and that would pose no problem whatsoever:

```
std::shared_ptr<Widget> spw2(spw1);      // spw2 uses same
                                          // control block as spw1
```

An especially surprising way that using raw pointer variables as std::shared_ptrconstructor arguments can lead to multiple control blocks involves the this pointer. Suppose our program uses std::shared_ptrs to manage Widget objects, and we have a data structure that keeps track of Widgets that have been processed:

```
std::vector<std::shared_ptr<Widget>> processedWidgets;
```

Further suppose that Widget has a member function that does the processing:

```
class Widget {
public:
  …
  void process();
  …
};
```

Here's a reasonable-looking approach for `Widget::process`:

```
void Widget::process()
{
  …                                      // process the Widget

  processedWidgets.emplace_back(this);   // add it to list of
}                                        // processed Widgets;
                                         // this is wrong!
```

The comment about this being wrong says it all—or at least most of it. (The part that's wrong is the passing of `this`, not the use of `emplace_back`. If you're not familiar with `emplace_back`, see Item 42.) This code will compile, but it's passing a raw pointer (`this`) to a container of `std::shared_ptr`s. The `std::shared_ptr` thus constructed will create a new control block for the pointed-to `Widget` (`*this`). That doesn't sound harmful until you realize that if there are `std::shared_ptr`s outside the member function that already point to that `Widget`, it's game, set, and match for undefined behavior.

The `std::shared_ptr` API includes a facility for just this kind of situation. It has probably the oddest of all names in the Standard C++ Library: `std::enable_shared_from_this`. That's a template for a base class you inherit from if you want a class managed by `std::shared_ptr`s to be able to safely create a `std::shared_ptr` from a `this` pointer. In our example, `Widget` would inherit from `std::enable_shared_from_this` as follows:

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
  …
  void process();
  …
};
```

As I said, `std::enable_shared_from_this` is a base class template. Its type parameter is always the name of the class being derived, so `Widget` inherits from `std::enable_shared_from_this<Widget>`.

If the idea of a derived class inheriting from a base class templatized on the derived class makes your head hurt, try not to think about it. The code is completely legal, and the design pattern behind it is so well established, it has a standard name, albeit one that's almost as odd as `std::enable_shared_from_this`. The name is *The Curiously Recurring Template Pattern (CRTP)*. If you'd like to learn more about it, unleash your search engine, because here we need to get back to `std::enable_shared_from_this`.

`std::enable_shared_from_this` defines a member function that creates a `std::shared_ptr` to the current object, but it does it without duplicating control blocks. The member function is `shared_from_this`, and you use it in member functions whenever you want a `std::shared_ptr` that points to the same object as the `this`pointer. Here's a safe implementation of `Widget::process`:

```
void Widget::process()
{
  // as before, process the Widget
  …

  // add std::shared_ptr to current object to processedWidgets
  processedWidgets.emplace_back(shared_from_this());
}
```

Internally, `shared_from_this` looks up the control block for the current object, and it creates a new `std::shared_ptr` that refers to that control block. The design relies on the current object having an associated control block. For that to be the case, there must be an existing `std::shared_ptr` (e.g., one outside the member function calling `shared_from_this`) that points to the current object. If no such `std::shared_ptr` exists (i.e., if the current object has no associated control block), behavior is undefined, although `shared_from_this` typically throws an exception.

To prevent clients from calling member functions that invoke `shared_from_this` before a `std::shared_ptr` points to the object, classes inheriting from `std::enable_shared_from_this` often declare their constructors `private` and have clients create objects by calling factory functions that return `std::shared_ptrs`. `Widget`, for example, could look like this:

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
  // factory function that perfect-forwards args
  // to a private ctor
  template<typename... Ts>
```

```
    static std::shared_ptr<Widget> create(Ts&&... params);

    …
    void process();             // as before
    …

  private:
    …                           // ctors
  };
```

By now, you may only dimly recall that our discussion of control blocks was motivated by a desire to understand the costs associated with `std::shared_ptr`s. Now that we understand how to avoid creating too many control blocks, let's return to the original topic.

A control block is typically only a few words in size, although custom deleters and allocators may make it larger. The usual control block implementation is more sophisticated than you might expect. It makes use of inheritance, and there's even a virtual function. (It's used to ensure that the pointed-to object is properly destroyed.) That means that using `std::shared_ptr`s also incurs the cost of the machinery for the virtual function used by the control block.

Having read about dynamically allocated control blocks, arbitrarily large deleters and allocators, virtual function machinery, and atomic reference count manipulations, your enthusiasm for `std::shared_ptr`s may have waned somewhat. That's fine. They're not the best solution to every resource management problem. But for the functionality they provide, `std::shared_ptr`s exact a very reasonable cost. Under typical conditions, where the default deleter and default allocator are used and where the `std::shared_ptr` is created by `std::make_shared`, the control block is only about three words in size, and its allocation is essentially free. (It's incorporated into the memory allocation for the object being pointed to. For details, see Item 21.) Dereferencing a `std::shared_ptr` is no more expensive than dereferencing a raw pointer. Performing an operation requiring a reference count manipulation (e.g., copy construction, assignment, destruction) entails one or two atomic operations, but these operations typically map to individual machine instructions, so although they may be expensive compared to non-atomic instructions, they're still just single instructions. The virtual function machinery in the control block is generally used only once per object managed by `std::shared_ptr`s: when the object is destroyed.

In exchange for these rather modest costs, you get automatic lifetime management of dynamically allocated resources. Most of the time, using `std::shared_ptr` is vastly preferable to trying to manage the lifetime of an object with shared ownership by hand. If you find yourself doubting whether you can afford use of `std::shared_ptr`, reconsider whether you really need shared ownership. If exclusive ownership will do or even *may* do, `std::unique_ptr` is a better choice. Its performance profile is close to that for raw pointers, and "upgrading" from `std::unique_ptr` to `std::shared_ptr` is easy, because a `std::shared_ptr` can be created from a `std::unique_ptr`.

The reverse is not true. Once you've turned lifetime management of a resource over to a `std::shared_ptr`, there's no changing your mind. Even if the reference count is one, you can't reclaim ownership of the resource in order to, say, have a `std::unique_ptr` manage it. The ownership contract between a resource and the `std::shared_ptr`s that point to it is of the 'til-death-do-us-part variety. No divorce, no annulment, no dispensations.

Something else `std::shared_ptr`s can't do is work with arrays. In yet another difference from `std::unique_ptr`, `std::shared_ptr` has an API that's designed only for pointers to single objects. There's no `std::shared_ptr<T[]>`. From time to time, "clever" programmers stumble on the idea of using a `std::shared_ptr<T>` to point to an array, specifying a custom deleter to perform an array delete (i.e., `delete []`). This can be made to compile, but it's a horrible idea. For one thing, `std::shared_ptr` offers no `operator[]`, so indexing into the array requires awkward expressions based on pointer arithmetic. For another, `std::shared_ptr` supports derived-to-base pointer conversions that make sense for single objects, but that open holes in the type system when applied to arrays. (For this reason, the `std::unique_ptr<T[]>` API prohibits such conversions.) Most importantly, given the variety of C++11 alternatives to built-in arrays (e.g., `std::array`, `std::vector`, `std::string`), declaring a smart pointer to a dumb array is almost always a sign of bad design.

Things to Remember

- `std::shared_ptr`s offer convenience approaching that of garbage collection for the shared lifetime management of arbitrary resources.
- Compared to `std::unique_ptr`, `std::shared_ptr` objects are typically twice as big, incur overhead for control blocks, and require atomic reference count manipulations.
- Default resource destruction is via `delete`, but custom deleters are supported. The type of the deleter has no effect on the type of the `std::shared_ptr`.
- Avoid creating `std::shared_ptr`s from variables of raw pointer type.

Item 20: Use `std::weak_ptr` for `std::shared_ptr`-like pointers that can dangle.

Paradoxically, it can be convenient to have a smart pointer that acts like a `std::shared_ptr` (see Item 19), but that doesn't participate in the shared ownership of the pointed-to resource. In other words, a pointer like `std::shared_ptr` that doesn't affect an object's reference count. This kind of smart pointer has to contend with a problem unknown to `std::shared_ptrs`: the possibility that what it points to has been destroyed. A truly smart pointer would deal with this problem by tracking when it *dangles*, i.e., when the object it is supposed to point to no longer exists. That's precisely the kind of smart pointer `std::weak_ptr` is.

You may be wondering how a `std::weak_ptr` could be useful. You'll probably wonder even more when you examine the `std::weak_ptr` API. It looks anything but smart. `std::weak_ptrs` can't be dereferenced, nor can they be tested for nullness. That's because `std::weak_ptr` isn't a standalone smart pointer. It's an augmentation of `std::shared_ptr`.

The relationship begins at birth. `std::weak_ptrs` are typically created from `std::shared_ptrs`. They point to the same place as the `std::shared_ptrs` initializing them, but they don't affect the reference count of the object they point to:

```
auto spw =                       // after spw is constructed,
  std::make_shared<Widget>();    // the pointed-to Widget's
                                 // ref count (RC) is 1. (See
                                 // Item 21 for info on
                                 // std::make_shared.)
…

std::weak_ptr<Widget> wpw(spw);  // wpw points to same Widget
                                 // as spw. RC remains 1
…

spw = nullptr;                   // RC goes to 0, and the
                                 // Widget is destroyed.
                                 // wpw now dangles
```

`std::weak_ptrs` that dangle are said to have *expired*. You can test for this directly,

```
if (wpw.expired()) …            // if wpw doesn't point
                                 // to an object…
```

but often what you desire is a check to see if a `std::weak_ptr` has expired and, if it hasn't (i.e., if it's not dangling), to access the object it points to. This is easier desired than done. Because `std::weak_ptrs` lack dereferencing operations, there's no way to write the code. Even if there were, separating the check and the dereference would introduce a race condition: between the call to `expired` and the dereferencing action, another thread might reassign or destroy the last `std::shared_ptr` pointing to the object, thus causing that object to be destroyed. In that case, your dereference would yield undefined behavior.

What you need is an atomic operation that checks to see if the `std::weak_ptr` has expired and, if not, gives you access to the object it points to. This is done by creating a `std::shared_ptr` from the `std::weak_ptr`. The operation comes in two forms, depending on what you'd like to have happen if the `std::weak_ptr` has expired when you try to create a `std::shared_ptr` from it. One form is `std::weak_ptr::lock`, which returns a `std::shared_ptr`. The `std::shared_ptr` is null if the `std::weak_ptr` has expired:

```
std::shared_ptr<Widget> spw1 = wpw.lock();  // if wpw's expired,
                                             // spw1 is null

auto spw2 = wpw.lock();                      // same as above,
                                             // but uses auto
```

The other form is the `std::shared_ptr` constructor taking a `std::weak_ptr` as an argument. In this case, if the `std::weak_ptr` has expired, an exception is thrown:

```
std::shared_ptr<Widget> spw3(wpw);    // if wpw's expired,
                                       // throw std::bad_weak_ptr
```

But you're probably still wondering about how `std::weak_ptr`s can be useful. Consider a factory function that produces smart pointers to read-only objects based on a unique ID. In accord with Item 18's advice regarding factory function return types, it returns a `std::unique_ptr`:

```
std::unique_ptr<const Widget> loadWidget(WidgetID id);
```

If `loadWidget` is an expensive call (e.g., because it performs file or database I/O) and it's common for IDs to be used repeatedly, a reasonable optimization would be to write a function that does what `loadWidget` does, but also caches its results. Clogging the cache with every `Widget` that has ever been requested can lead to performance problems of its own, however, so another reasonable optimization would be to destroy cached `Widget`s when they're no longer in use.

For this caching factory function, a `std::unique_ptr` return type is not a good fit. Callers should certainly receive smart pointers to cached objects, and callers should certainly determine the lifetime of those objects, but the cache needs a pointer to the objects, too. The cache's pointers need to be able to detect when they dangle, because when factory clients are finished using an object returned by the factory, that object will be destroyed, and the corresponding cache entry will dangle. The cached pointers should therefore be `std::weak_ptrs`—pointers that can detect when they dangle. That means that the factory's return type should be a `std::shared_ptr`, because `std::weak_ptrs` can detect when they dangle only when an object's lifetime is managed by `std::shared_ptrs`.

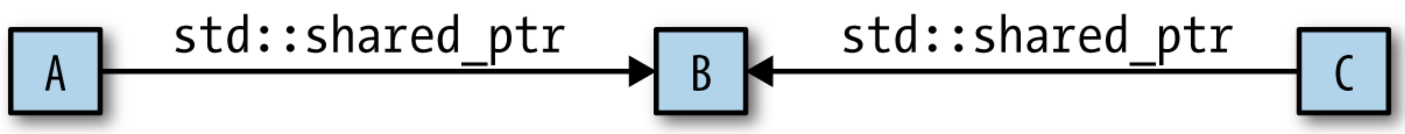
Here's a quick-and-dirty implementation of a caching version of `loadWidget`:

```
std::shared_ptr<const Widget> fastLoadWidget(WidgetID id)
{
  static std::unordered_map<WidgetID,
                            std::weak_ptr<const Widget>> cache;

  auto objPtr = cache[id].lock();   // objPtr is std::shared_ptr
                                    // to cached object (or null
                                    // if object's not in cache)

  if (!objPtr) {                    // if not in cache,
    objPtr = loadWidget(id);        // load it
    cache[id] = objPtr;             // cache it
  }
  return objPtr;
}
```

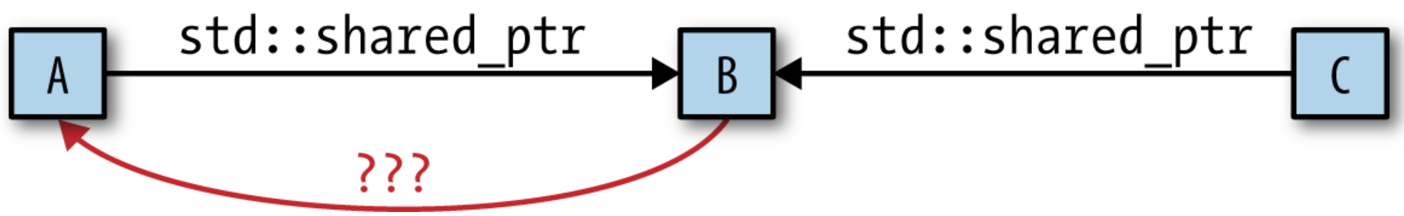
This implementation employs one of C++11's hash table containers (`std::unordered_map`), though it doesn't show the `WidgetID` hashing and equality-comparison functions that would also have to be present.

The implementation of `fastLoadWidget` ignores the fact that the cache may accumulate expired `std::weak_ptrs` corresponding to `Widgets` that are no longer in use (and have therefore been destroyed). The implementation can be refined, but rather than spend time on an issue that lends no additional insight into `std::weak_ptrs`, let's consider a second use case: the Observer design pattern. The primary components of this pattern are subjects (objects whose state may change) and observers (objects to be notified when state changes occur). In most implementations, each subject contains a data member holding pointers to its observers. That makes it easy for subjects to issue state change notifications. Subjects have no interest in controlling the lifetime of their observers (i.e., when they're destroyed), but they have a great interest in making sure that if an observer gets destroyed, subjects don't try to subsequently access it. A reasonable design is for each subject to hold a container of `std::weak_ptrs` to its observers, thus making it possible for the subject to determine whether a pointer dangles before using it.

As a final example of `std::weak_ptr`'s utility, consider a data structure with objects `A`, `B`, and `C` in it, where `A` and `C` share ownership of `B` and therefore hold `std::shared_ptr`s to it:



Suppose it'd be useful to also have a pointer from `B` back to `A`. What kind of pointer should this be?



There are three choices:

- **A raw pointer.** With this approach, if `A` is destroyed, but `C` continues to point to `B`, `B`will contain a pointer to `A` that will dangle. `B` won't be able to detect that, so `B` may inadvertently dereference the dangling pointer. That would yield undefined behavior.
- **A `std::shared_ptr`.** In this design, `A` and `B` contain `std::shared_ptr`s to each other. The resulting `std::shared_ptr` cycle (`A` points to `B` and `B` points to `A`) will prevent both `A` and `B` from being destroyed. Even if `A` and `B` are unreachable from other program data structures (e.g., because `C` no longer points to `B`), each will have a reference count of one. If that happens, `A` and `B` will have been leaked, for all practical purposes: it will be impossible for the program to access them, yet their resources will never be reclaimed.
- **A `std::weak_ptr`.** This avoids both problems above. If `A` is destroyed, `B`'s pointer back to it will dangle, but `B` will be able to detect that. Furthermore, though `A` and `B` will point to one another, `B`'s pointer won't affect `A`'s reference count, hence can't keep `A`from being destroyed when `std::shared_ptr`s no longer point to it.

Using `std::weak_ptr` is clearly the best of these choices. However, it's worth noting that the need to employ `std::weak_ptr`s to break prospective cycles of `std::shared_ptr`s is not terribly common. In strictly hierarchal data structures such as trees, child nodes are typically owned only by their parents.

When a parent node is destroyed, its child nodes should be destroyed, too. Links from parents to children are thus generally best represented by `std::unique_ptrs`. Back-links from children to parents can be safely implemented as raw pointers, because a child node should never have a lifetime longer than its parent. There's thus no risk of a child node dereferencing a dangling parent pointer.

Of course, not all pointer-based data structures are strictly hierarchical, and when that's the case, as well as in situations such as caching and the implementation of lists of observers, it's nice to know that `std::weak_ptr` stands at the ready.

From an efficiency perspective, the `std::weak_ptr` story is essentially the same as that for `std::shared_ptr`. `std::weak_ptr` objects are the same size as `std::shared_ptr`objects, they make use of the same control blocks as `std::shared_ptrs` (see Item 19), and operations such as construction, destruction, and assignment involve atomic reference count manipulations. That probably surprises you, because I wrote at the beginning of this Item that `std::weak_ptrs` don't participate in reference counting. Except that's not quite what I wrote. What I wrote was that `std::weak_ptrs` don't participate in the *shared ownership* of objects and hence don't affect the *pointed-to object's reference count*. There's actually a second reference count in the control block, and it's this second reference count that `std::weak_ptrs` manipulate. For details, continue on to Item 21.

Things to Remember

- Use `std::weak_ptr` for `std::shared_ptr`-like pointers that can dangle.
- Potential use cases for `std::weak_ptr` include caching, observer lists, and the prevention of `std::shared_ptr` cycles.

Item 21: Prefer `std::make_unique` and `std::make_shared` to direct use of `new`.

Let's begin by leveling the playing field for `std::make_unique` and `std::make_shared`. `std::make_shared` is part of C++11, but, sadly, `std::make_unique` isn't. It joined the Standard Library as of C++14. If you're using C++11, never fear, because a basic version of `std::make_unique` is easy to write yourself. Here, look:

```
template<typename T, typename... Ts>
std::unique_ptr<T> make_unique(Ts&&... params)
```

```
{
  return std::unique_ptr<T>(new T(std::forward<Ts>(params)...));
}
```

As you can see, make_unique just perfect-forwards its parameters to the constructor of the object being created, constructs a std::unique_ptr from the raw pointer new produces, and returns the std::unique_ptr so created. This form of the function doesn't support arrays (see Item 18), but it demonstrates that with only a little effort, you can create make_unique if you need to.[3] Just remember not to put your version in namespace std, because you won't want it to clash with a vendor-provided version when you upgrade to a C++14 Standard Library implementation.

std::make_unique and std::make_shared are two of the three *make functions*: functions that take an arbitrary set of arguments, perfect-forward them to the constructor for a dynamically allocated object, and return a smart pointer to that object. The third make function is std::allocate_shared. It acts just like std::make_shared, except its first argument is an allocator object to be used for the dynamic memory allocation.

Even the most trivial comparison of smart pointer creation using and not using a makefunction reveals the first reason why using such functions is preferable. Consider:

```
auto upw1(std::make_unique<Widget>());      // with make func

std::unique_ptr<Widget> upw2(new Widget);   // without make func


auto spw1(std::make_shared<Widget>());      // with make func

std::shared_ptr<Widget> spw2(new Widget);   // without make func
```

I've highlighted the essential difference: the versions using new repeat the type being created, but the make functions don't. Repeating types runs afoul of a key tenet of software engineering: code duplication should be avoided. Duplication in source code increases compilation times, can lead to bloated object code, and generally renders a code base more difficult to work with. It often evolves into inconsistent code, and inconsistency in a code base often leads to bugs. Besides, typing something twice takes more effort than typing it once, and who's not a fan of reducing their typing burden?

The second reason to prefer make functions has to do with exception safety. Suppose we have a function to process a Widget in accord with some priority:

```
void processWidget(std::shared_ptr<Widget> spw, int priority);
```

Passing the `std::shared_ptr` by value may look suspicious, but Item 41 explains that if `processWidget` always makes a copy of the `std::shared_ptr` (e.g., by storing it in a data structure tracking `Widgets` that have been processed), this can be a reasonable design choice.

Now suppose we have a function to compute the relevant priority,

```
int computePriority();
```

and we use that in a call to `processWidget` that uses `new` instead of `std::make_shared`:

```
processWidget(std::shared_ptr<Widget>(new Widget),   // potential
              computePriority());                    // resource
                                                     // leak!
```

As the comment indicates, this code could leak the `Widget` conjured up by `new`. But how? Both the calling code and the called function are using `std::shared_ptrs`, and `std::shared_ptrs` are designed to prevent resource leaks. They automatically destroy what they point to when the last `std::shared_ptr` pointing there goes away. If everybody is using `std::shared_ptrs` everywhere, how can this code leak?

The answer has to do with compilers' translation of source code into object code. At runtime, the arguments for a function must be evaluated before the function can be invoked, so in the call to `processWidget`, the following things must occur before `processWidget` can begin execution:

- The expression "`new Widget`" must be evaluated, i.e., a `Widget` must be created on the heap.
- The constructor for the `std::shared_ptr<Widget>` responsible for managing the pointer produced by `new` must be executed.
- `computePriority` must run.

Compilers are not required to generate code that executes them in this order. "`newWidget`" must be executed before the `std::shared_ptr` constructor may be called, because the result of that `new` is used as an argument to that constructor, but `computePriority` may be executed before those calls, after them, or, crucially, *between*them. That is, compilers may emit code to execute the operations in this order:

1. Perform “`new Widget`”.
2. Execute `computePriority`.
3. Run `std::shared_ptr` constructor.

If such code is generated and, at runtime, `computePriority` produces an exception, the dynamically allocated `Widget` from Step 1 will be leaked, because it will never be stored in the `std::shared_ptr` that’s supposed to start managing it in Step 3.

Using `std::make_shared` avoids this problem. Calling code would look like this:

```
processWidget(std::make_shared<Widget>(),    // no potential
              computePriority());             // resource leak
```

At runtime, either `std::make_shared` or `computePriority` will be called first. If it’s `std::make_shared`, the raw pointer to the dynamically allocated `Widget` is safely stored in the returned `std::shared_ptr` before `computePriority` is called. If `computePriority` then yields an exception, the `std::shared_ptr` destructor will see to it that the `Widget` it owns is destroyed. And if `computePriority` is called first and yields an exception, `std::make_shared` will not be invoked, and there will hence be no dynamically allocated `Widget` to worry about.

If we replace `std::shared_ptr` and `std::make_shared` with `std::unique_ptr` and `std::make_unique`, exactly the same reasoning applies. Using `std::make_unique`instead of `new` is thus just as important in writing exception-safe code as using `std::make_shared`.

A special feature of `std::make_shared` (compared to direct use of `new`) is improved efficiency. Using `std::make_shared` allows compilers to generate smaller, faster code that employs leaner data structures. Consider the following direct use of `new`:

```
std::shared_ptr<Widget> spw(new Widget);
```

It’s obvious that this code entails a memory allocation, but it actually performs two. Item 19 explains that every `std::shared_ptr` points to a control block containing, among other things, the reference count for the pointed-to object. Memory for this control block is allocated in the `std::shared_ptr` constructor. Direct use of `new`, then, requires one memory allocation for the `Widget` and a second allocation for the control block.

If `std::make_shared` is used instead,

```
auto spw = std::make_shared<Widget>();
```

one allocation suffices. That's because `std::make_shared` allocates a single chunk of memory to hold both the `Widget` object and the control block. This optimization reduces the static size of the program, because the code contains only one memory allocation call, and it increases the speed of the executable code, because memory is allocated only once. Furthermore, using `std::make_shared` obviates the need for some of the bookkeeping information in the control block, potentially reducing the total memory footprint for the program.

The efficiency analysis for `std::make_shared` is equally applicable to `std::allocate_shared`, so the performance advantages of `std::make_shared` extend to that function, as well.

The arguments for preferring `make` functions over direct use of `new` are strong ones. Despite their software engineering, exception safety, and efficiency advantages, however, this Item's guidance is to *prefer* the `make` functions, not to rely on them exclusively. That's because there are circumstances where they can't or shouldn't be used.

For example, none of the `make` functions permit the specification of custom deleters (see Items 18 and 19), but both `std::unique_ptr` and `std::shared_ptr` have constructors that do. Given a custom deleter for a `Widget`,

```
auto widgetDeleter = [](Widget* pw) { … };
```

creating a smart pointer using it is straightforward using `new`:

```
std::unique_ptr<Widget, decltype(widgetDeleter)>
  upw(new Widget, widgetDeleter);

std::shared_ptr<Widget> spw(new Widget, widgetDeleter);
```

There's no way to do the same thing with a `make` function.

A second limitation of `make` functions stems from a syntactic detail of their implementations. Item 7 explains that when creating an object whose type overloads constructors both with and without `std::initializer_list` parameters, creating the object using braces prefers the `std::initializer_list` constructor, while creating the object using parentheses calls the nonstd::initializer_list constructor. The makefunctions perfect-forward their parameters to an object's constructor, but do they do so using parentheses or using braces? For some types, the answer to this question makes a big difference. For example, in these calls,

```
auto upv = std::make_unique<std::vector<int>>(10, 20);

auto spv = std::make_shared<std::vector<int>>(10, 20);
```

do the resulting smart pointers point to std::vectors with 10 elements, each of value 20, or to std::vectors with two elements, one with value 10 and the other with value 20? Or is the result indeterminate?

The good news is that it's not indeterminate: both calls create std::vectors of size 10 with all values set to 20. That means that within the make functions, the perfect forwarding code uses parentheses, not braces. The bad news is that if you want to construct your pointed-to object using a braced initializer, you must use new directly. Using a makefunction would require the ability to perfect-forward a braced initializer, but, as Item 30explains, braced initializers can't be perfect-forwarded. However, Item 30 also describes a workaround: use auto type deduction to create a std::initializer_list object from a braced initializer (see Item 2), then pass the auto-created object through the makefunction:

```
// create std::initializer_list
auto initList = { 10, 20 };

// create std::vector using std::initializer_list ctor
auto spv = std::make_shared<std::vector<int>>(initList);
```

For std::unique_ptr, these two scenarios (custom deleters and braced initializers) are the only ones where its make functions are problematic. For std::shared_ptr and its make functions, there are two more. Both are edge cases, but some developers live on the edge, and you may be one of them.

Some classes define their own versions of operator new and operator delete. The presence of these functions implies that the global memory allocation and deallocation routines for objects of these types are inappropriate. Often, class-specific routines are designed only to allocate and deallocate chunks of memory of precisely the size of objects of the class, e.g., operator new and operator delete for class Widget are often designed only to handle allocation and deallocation of chunks of memory of exactly size sizeof(Widget). Such routines are a poor fit for std::shared_ptr's support for custom allocation (via std::allocate_shared) and deallocation (via custom deleters), because the amount of memory that `std::allocate_shared` requests isn't the size of the dynamically allocated object, it's the size of that object *plus* the size of a control block. Consequently, using `make` functions to create objects of types with class-specific versions of `operator new` and `operator delete` is typically a poor idea.

The size and speed advantages of `std::make_shared` vis-à-vis direct use of `new` stem from `std::shared_ptr`'s control block being placed in the same chunk of memory as the managed object. When that object's reference count goes to zero, the object is destroyed (i.e., its destructor is called). However, the memory it occupies can't be released until the control block has also been destroyed, because the same chunk of dynamically allocated memory contains both.

As I noted, the control block contains bookkeeping information beyond just the reference count itself. The reference count tracks how many `std::shared_ptr`s refer to the control block, but the control block contains a second reference count, one that tallies how many `std::weak_ptr`s refer to the control block. This second reference count is known as the *weak count*.[4] When a `std::weak_ptr` checks to see if it has expired (see Item 19), it does so by examining the reference count (not the weak count) in the control block that it refers to. If the reference count is zero (i.e., if the pointed-to object has no `std::shared_ptr`s referring to it and has thus been destroyed), the `std::weak_ptr` has expired. Otherwise, it hasn't.

As long as `std::weak_ptr`s refer to a control block (i.e., the weak count is greater than zero), that control block must continue to exist. And as long as a control block exists, the memory containing it must remain allocated. The memory allocated by a `std::shared_ptr` `make` function, then, can't be deallocated until the last `std::shared_ptr` *and* the last `std::weak_ptr` referring to it have been destroyed.

If the object type is quite large and the time between destruction of the last `std::shared_ptr` and the last `std::weak_ptr` is significant, a lag can occur between when an object is destroyed and when the memory it occupied is freed:

```
class ReallyBigType { … };

auto pBigObj =                          // create very large
  std::make_shared<ReallyBigType>();    // object via
                                        // std::make_shared

…             // create std::shared_ptrs and std::weak_ptrs to
              // large object, use them to work with it

…             // final std::shared_ptr to object destroyed here,
              // but std::weak_ptrs to it remain
```

```
…             // during this period, memory formerly occupied
              // by large object remains allocated

…             // final std::weak_ptr to object destroyed here;
              // memory for control block and object is released
```

With a direct use of new, the memory for the ReallyBigType object can be released as soon as the last std::shared_ptr to it is destroyed:

```
class ReallyBigType { … };              // as before

std::shared_ptr<ReallyBigType> pBigObj(new ReallyBigType);
                                        // create very large
                                        // object via new

…             // as before, create std::shared_ptrs and
              // std::weak_ptrs to object, use them with it

…             // final std::shared_ptr to object destroyed here,
              // but std::weak_ptrs to it remain;
              // memory for object is deallocated

…             // during this period, only memory for the
              // control block remains allocated

…             // final std::weak_ptr to object destroyed here;
              // memory for control block is released
```

Should you find yourself in a situation where use of std::make_shared is impossible or inappropriate, you'll want to guard yourself against the kind of exception-safety problems we saw earlier. The best way to do that is to make sure that when you use new directly, you immediately pass the result to a smart pointer constructor in *a statement that does nothing else*. This prevents compilers from generating code that could emit an exception between the use of new and invocation of the constructor for the smart pointer that will manage the newed object.

As an example, consider a minor revision to the exception-unsafe call to the processWidget function we examined earlier. This time, we'll specify a custom deleter:

```
void processWidget(std::shared_ptr<Widget> spw,  // as before
                   int priority);

void cusDel(Widget *ptr);                        // custom
                                                 // deleter
```

Here's the exception-unsafe call:

```
processWidget(                                   // as before,
  std::shared_ptr<Widget>(new Widget, cusDel),   // potential
  computePriority()                              // resource
);                                               // leak!
```

Recall: if `computePriority` is called after "`new Widget`" but before the `std::shared_ptr`constructor, and if `computePriority` yields an exception, the dynamically allocated `Widget` will be leaked.

Here the use of a custom deleter precludes use of `std::make_shared`, so the way to avoid the problem is to put the allocation of the `Widget` and the construction of the `std::shared_ptr` into their own statement, then call `processWidget` with the resulting `std::shared_ptr`. Here's the essence of the technique, though, as we'll see in a moment, we can tweak it to improve its performance:

```
std::shared_ptr<Widget> spw(new Widget, cusDel);

processWidget(spw, computePriority());     // correct, but not
                                           // optimal; see below
```

This works, because a `std::shared_ptr` assumes ownership of the raw pointer passed to its constructor, even if that constructor yields an exception. In this example, if `spw`'s constructor throws an exception (e.g., due to an inability to dynamically allocate memory for a control block), it's still guaranteed that `cusDel` will be invoked on the pointer resulting from "`new Widget`".

The minor performance hitch is that in the exception-unsafe call, we're passing an rvalue to `processWidget`,

```
processWidget(
  std::shared_ptr<Widget>(new Widget, cusDel),  // arg is rvalue
  computePriority()
);
```

but in the exception-safe call, we're passing an lvalue:

```
processWidget(spw, computePriority());         // arg is lvalue
```

Because `processWidget`'s `std::shared_ptr` parameter is passed by value, construction from an rvalue entails only a move, while construction from an lvalue requires a copy. For `std::shared_ptr`, the difference can be significant, because copying a `std::shared_ptr`requires an atomic increment of its reference count, while moving a `std::shared_ptr`requires no reference count manipulation at all. For the exception-safe code to achieve the level of performance of the exception-unsafe code, we need to apply `std::move` to `spw` to turn it into an rvalue (see Item 23):

```
processWidget(std::move(spw),            // both efficient and
              computePriority());         // exception safe
```

That's interesting and worth knowing, but it's also typically irrelevant, because you'll rarely have a reason not to use a `make` function. And unless you have a compelling reason for doing otherwise, using a `make` function is what you should do.

Things to Remember

- Compared to direct use of `new`, `make` functions eliminate source code duplication, improve exception safety, and, for `std::make_shared` and `std::allocate_shared`, generate code that's smaller and faster.
- Situations where use of `make` functions is inappropriate include the need to specify custom deleters and a desire to pass braced initializers.
- For `std::shared_ptrs`, additional situations where `make` functions may be ill-advised include (1) classes with custom memory management and (2) systems with memory concerns, very large objects, and `std::weak_ptrs` that outlive the corresponding `std::shared_ptrs`.

Item 22: When using the Pimpl Idiom, define special member functions in the implementation file.

If you've ever had to combat excessive build times, you're familiar with the *Pimpl* ("pointer to implementation") *Idiom*. That's the technique whereby you replace the data members of a class with a pointer to an implementation class (or struct), put the data members that used to be in the primary class into the implementation class, and access those data members indirectly through the pointer. For example, suppose `Widget` looks like this:

```
class Widget {                     // in header "widget.h"
public:
  Widget();
  …
private:
  std::string name;
  std::vector<double> data;
  Gadget g1, g2, g3;               // Gadget is some user-
};                                 // defined type
```

Because `Widget`'s data members are of types `std::string`, `std::vector`, and `Gadget`, headers for those types must be present for `Widget` to compile, and that means that `Widget` clients must `#include <string>`, `<vector>`, and `gadget.h`. Those headers increase the compilation time for `Widget` clients, plus they make those clients dependent on the contents of the headers. If a header's content changes, `Widget` clients must recompile. The standard headers `<string>` and `<vector>` don't change very often, but it could be that `gadget.h` is subject to frequent revision.

Applying the Pimpl Idiom in C++98 could have `Widget` replace its data members with a raw pointer to a struct that has been declared, but not defined:

```
class Widget {                 // still in header "widget.h"
public:
  Widget();
  ~Widget();                   // dtor is needed—see below
  …

private:
  struct Impl;                // declare implementation struct
  Impl *pImpl;                // and pointer to it
};
```

Because `Widget` no longer mentions the types `std::string`, `std::vector`, and `Gadget`, `Widget` clients no longer need to `#include` the headers for these types. That speeds compilation, and it also means that if something in these headers changes, `Widget` clients are unaffected.

A type that has been declared, but not defined, is known as an *incomplete type*. `Widget::Impl` is such a type. There are very few things you can do with an incomplete type, but declaring a pointer to it is one of them. The Pimpl Idiom takes advantage of that.

Part 1 of the Pimpl Idiom is the declaration of a data member that's a pointer to an incomplete type. Part 2 is the dynamic allocation and deallocation of the object that holds the data members that used to be in the original class. The allocation and deallocation code goes in the implementation file, e.g., for `Widget`, in `widget.cpp`:

```
#include "widget.h"              // in impl. file "widget.cpp"
#include "gadget.h"
#include <string>
#include <vector>

struct Widget::Impl {            // definition of Widget::Impl
  std::string name;              // with data members formerly
  std::vector<double> data;      // in Widget
  Gadget g1, g2, g3;
};

Widget::Widget()                 // allocate data members for
: pImpl(new Impl)               // this Widget object
{}

Widget::~Widget()                // destroy data members for
{ delete pImpl; }               // this object
```

Here I'm showing `#include` directives to make clear that the overall dependencies on the headers for `std::string`, `std::vector`, and `Gadget` continue to exist. However, these dependencies have been moved from `widget.h` (which is visible to and used by `Widget`clients) to `widget.cpp` (which is visible to and used only by the `Widget` implementer). I've also highlighted the code that dynamically allocates and deallocates the `Impl` object. The need to deallocate this object when a `Widget` is destroyed is what necessitates the `Widget`destructor.

But I've shown you C++98 code, and that reeks of a bygone millennium. It uses raw pointers and raw `new` and raw `delete` and it's all just so…raw. This chapter is built on the idea that smart pointers are preferable to raw pointers, and if what we want is to dynamically allocate a `Widget::Impl` object inside the `Widget` constructor and have it destroyed at the same time the `Widget` is, `std::unique_ptr` (see Item 18) is precisely the tool we need. Replacing the raw `pImpl` pointer with a `std::unique_ptr` yields this code for the header file,

```
class Widget {                      // in "widget.h"
public:
  Widget();
  …

private:
  struct Impl;
```

```
    std::unique_ptr<Impl> pImpl;      // use smart pointer
};                                    // instead of raw pointer
```

and this for the implementation file:

```
#include "widget.h"                   // in "widget.cpp"
#include "gadget.h"
#include <string>
#include <vector>

struct Widget::Impl {                 // as before
  std::string name;
  std::vector<double> data;
  Gadget g1, g2, g3;
};

Widget::Widget()                      // per Item 21, create
: pImpl(std::make_unique<Impl>())    // std::unique_ptr
{}                                    // via std::make_unique
```

You'll note that the `Widget` destructor is no longer present. That's because we have no code to put into it. `std::unique_ptr` automatically deletes what it points to when it (the `std::unique_ptr`) is destroyed, so we need not delete anything ourselves. That's one of the attractions of smart pointers: they eliminate the need for us to sully our hands with manual resource release.

This code compiles, but, alas, the most trivial client use doesn't:

```
#include "widget.h"

Widget w;                             // error!
```

The error message you receive depends on the compiler you're using, but the text generally mentions something about applying `sizeof` or `delete` to an incomplete type. Those operations aren't among the things you can do with such types.

This apparent failure of the Pimpl Idiom using `std::unique_ptr`s is alarming, because (1) `std::unique_ptr` is advertised as supporting incomplete types, and (2) the Pimpl Idiom is one of `std::unique_ptr`s most common use cases. Fortunately, getting the code to work is easy. All that's required is a basic understanding of the cause of the problem.

The issue arises due to the code that's generated when `w` is destroyed (e.g., goes out of scope). At that point, its destructor is called. In the class definition using `std::unique_ptr`, we didn't declare a destructor, because we didn't have any code to put into it. In accord with the usual rules for compiler-generated special member functions (see Item 17), the compiler generates a destructor for us. Within that destructor, the compiler inserts code to call the destructor for `Widget`'s data member `pImpl`. `pImpl` is a `std::unique_ptr<Widget::Impl>`, i.e., a `std::unique_ptr` using the default deleter. The default deleter is a function that uses `delete` on the raw pointer inside the `std::unique_ptr`. Prior to using `delete`, however, implementations typically have the default deleter employ C++11's `static_assert` to ensure that the raw pointer doesn't point to an incomplete type. When the compiler generates code for the destruction of the `Widget w`, then, it generally encounters a `static_assert` that fails, and that's usually what leads to the error message. This message is associated with the point where `w` is destroyed, because `Widget`'s destructor, like all compiler-generated special member functions, is implicitly `inline`. The message itself often refers to the line where `w` is created, because it's the source code explicitly creating the object that leads to its later implicit destruction.

To fix the problem, you just need to make sure that at the point where the code to destroy the `std::unique_ptr<Widget::Impl>` is generated, `Widget::Impl` is a complete type. The type becomes complete when its definition has been seen, and `Widget::Impl` is defined inside `widget.cpp`. The key to successful compilation, then, is to have the compiler see the body of `Widget`'s destructor (i.e., the place where the compiler will generate code to destroy the `std::unique_ptr` data member) only inside `widget.cpp`after `Widget::Impl` has been defined.

Arranging for that is simple. Declare `Widget`'s destructor in `widget.h`, but don't define it there:

```
class Widget {                    // as before, in "widget.h"
public:
  Widget();
  ~Widget();                     // declaration only
  …

private:                          // as before
  struct Impl;
  std::unique_ptr<Impl> pImpl;
};
```

Define it in `widget.cpp` after `Widget::Impl` has been defined:

```
#include "widget.h"               // as before, in "widget.cpp"
#include "gadget.h"
#include <string>
```

```
#include <vector>

struct Widget::Impl {              // as before, definition of
  std::string name;                // Widget::Impl
  std::vector<double> data;
  Gadget g1, g2, g3;
};

Widget::Widget()                   // as before
: pImpl(std::make_unique<Impl>())
{}

Widget::~Widget()                 // ~Widget definition
{}
```

This works well, and it requires the least typing, but if you want to emphasize that the compiler-generated destructor would do the right thing—that the only reason you declared it was to cause its definition to be generated in `Widget`'s implementation file, you can define the destructor body with "`= default`":

```
Widget::~Widget() = default;       // same effect as above
```

Classes using the Pimpl Idiom are natural candidates for move support, because compiler-generated move operations do exactly what's desired: perform a move on the underlying `std::unique_ptr`. As Item 17 explains, the declaration of a destructor in `Widget` prevents compilers from generating the move operations, so if you want move support, you must declare the functions yourself. Given that the compiler-generated versions would behave correctly, you're likely to be tempted to implement them as follows:

```
class Widget {                                          // still in
public:                                                 // "widget.h"
  Widget();
  ~Widget();

  Widget(Widget&& rhs) noexcept = default;             // right idea,
  Widget& operator=(Widget&& rhs) noexcept = default;  // wrong code!

  …

private:                                                // as before
  struct Impl;
  std::unique_ptr<Impl> pImpl;
};
```

This approach leads to the same kind of problem as declaring the class without a destructor, and for the same fundamental reason. The compiler-generated move assignment operator needs to destroy the object pointed to by `pImpl` before reassigning it, but in the `Widget` header file, `pImpl` points to an incomplete type. The situation is different for the move constructor. The problem there is that compilers must be able to generate code to destroy `pImpl` in the event that an exception arises inside the move constructor (even if the constructor is `noexcept`!), and destroying `pImpl` requires that `Impl` be complete.

Because the problem is the same as before, so is the fix—move the definition of the move operations into the implementation file:

```
class Widget {                               // still in "widget.h"
public:
  Widget();
  ~Widget();

  Widget(Widget&& rhs) noexcept;             // declarations
  Widget& operator=(Widget&& rhs) noexcept;  // only

  …

private:                                     // as before
  struct Impl;
  std::unique_ptr<Impl> pImpl;
};

#include <string>                            // as before,
…                                            // in "widget.cpp"

struct Widget::Impl { … };                   // as before

Widget::Widget()                             // as before
: pImpl(std::make_unique<Impl>())
{}

Widget::~Widget() = default;                 // as before

                                             // definitions
Widget::Widget(Widget&& rhs) noexcept = default;
Widget& Widget::operator=(Widget&& rhs) noexcept = default;
```

The Pimpl Idiom is a way to reduce compilation dependencies between a class's implementation and the class's clients, but, conceptually, use of the idiom doesn't change what the class represents. The original `Widget` class contained `std::string`, `std::vector`, and `Gadget` data members, and, assuming that `Gadget`s, like `std::string`s and `std::vector`s, can be copied, it would make sense for `Widget` to support the copy operations. We have to write these functions ourselves, because (1) compilers won't generate copy operations for classes with move-only types like `std::unique_ptr` and (2) even if they did, the generated functions would copy only the `std::unique_ptr` (i.e., perform a *shallow copy*), and we want to copy what the pointer points to (i.e., perform a *deep copy*).

In a ritual that is by now familiar, we declare the functions in the header file and implement them in the implementation file:

```
class Widget {                         // still in "widget.h"
public:
  …                                    // other funcs, as before

  Widget(const Widget& rhs);            // declarations
  Widget& operator=(const Widget& rhs);  // only

private:                                  // as before
  struct Impl;
  std::unique_ptr<Impl> pImpl;
};


#include "widget.h"                 // as before,
…                                   // in "widget.cpp"


struct Widget::Impl { … };          // as before

Widget::~Widget() = default;        // other funcs, as before

Widget::Widget(const Widget& rhs)              // copy ctor
: pImpl(nullptr)
{ if (rhs.pImpl) pImpl = std::make_unique<Impl>(*rhs.pImpl); }

Widget& Widget::operator=(const Widget& rhs)   // copy operator=
{
  if (!rhs.pImpl) pImpl.reset();
  else if (!pImpl) pImpl = std::make_unique<Impl>(*rhs.pImpl);
  else *pImpl = *rhs.pImpl;

  return *this;
}
```

The implementations are straightforward, though we must handle cases where the parameter `rhs` or, in the case of the copy assignment operator, `*this` has been moved from and thus contains a null `pImpl` pointer. In general, we take advantage of the fact that compilers will create the copy operations for Impl, and these operations will copy each field automatically. We thus implement Widget's copy operations by calling Widget::Impl's compiler-generated copy operations. In both functions, note that we still follow the advice of Item 21 to prefer use of std::make_unique over direct use of new.

For purposes of implementing the Pimpl Idiom, std::unique_ptr is the smart pointer to use, because the pImpl pointer inside an object (e.g., inside a Widget) has exclusive ownership of the corresponding implementation object (e.g., the Widget::Impl object). Still, it's interesting to note that if we were to use std::shared_ptr instead of std::unique_ptr for pImpl (i.e., if the values in an Impl struct could be shared by multiple Widgets), we'd find that the advice of this Item no longer applied. There'd be no need to declare a destructor in Widget, and without a user-declared destructor, compilers would happily generate the move operations, which would do exactly what we'd want them to. That is, given this code in widget.h,

```
class Widget {                       // in "widget.h"
public:
  Widget();
  …                                  // no declarations for dtor
                                     // or move operations
private:
  struct Impl;
  std::shared_ptr<Impl> pImpl;      // std::shared_ptr
};                                   // instead of std::unique_ptr
```

and this client code that #includes widget.h,

```
Widget w1;

auto w2(std::move(w1));              // move-construct w2

w1 = std::move(w2);                  // move-assign w1
```

everything would compile and run as we'd hope: w1 would be default constructed, its value would be moved into w2, that value would be moved back into w1, and then both w1 and w2would be destroyed (thus causing the pointed-to Widget::Impl object to be destroyed).

The difference in behavior between std::unique_ptr and std::shared_ptr for pImplpointers stems from the differing ways these smart pointers support custom deleters. For std::unique_ptr, the type of the deleter is part of the type of the smart pointer, and this makes it possible for compilers to generate smaller runtime data structures and faster runtime code. A consequence of this greater efficiency is that pointed-to types must be complete when compiler-generated special functions (e.g., destructors or move operations) are used. For `std::shared_ptr`, the type of the deleter is not part of the type of the smart pointer. This necessitates larger runtime data structures and somewhat slower code, but pointed-to types need not be complete when compiler-generated special functions are employed.

For the Pimpl Idiom, there's not really a trade-off between the characteristics of `std::unique_ptr` and `std::shared_ptr`, because the relationship between classes like `Widget` and classes like `Widget::Impl` is exclusive ownership, and that makes `std::unique_ptr` the proper tool for the job. Nevertheless, it's worth knowing that in other situations—situations where shared ownership exists (and `std::shared_ptr` is hence a fitting design choice), there's no need to jump through the function-definition hoops that use of `std::unique_ptr` entails.

Things to Remember

- The Pimpl Idiom decreases build times by reducing compilation dependencies between class clients and class implementations.
- For `std::unique_ptr pImpl` pointers, declare special member functions in the class header, but implement them in the implementation file. Do this even if the default function implementations are acceptable.
- The above advice applies to `std::unique_ptr`, but not to `std::shared_ptr`.

---

[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.

[2] This implementation is not required by the Standard, but every Standard Library implementation I'm familiar with employs it.

[3] To create a full-featured `make_unique` with the smallest effort possible, search for the standardization document that gave rise to it, then copy the implementation you'll find there. The

document you want is N3656 by Stephan T. Lavavej, dated 2013-04-18.

4 In practice, the value of the weak count isn't always equal to the number of `std::weak_ptr`s referring to the control block, because library implementers have found ways to slip additional information into the weak count that facilitate better code generation. For purposes of this Item, we'll ignore this and assume that the weak count's value is the number of `std::weak_ptr`s referring to the control block.