7

Smart Pointers

Hecatombs of code and rivers of ink have been dedicated to smart pointers by programmers and writers around the world. Perhaps the most popular, intricate, and powerful C++ idiom, smart pointers are interesting in that they combine many syntactic and semantic issues. This chapter discusses smart pointers, from their simplest aspects to their most complex ones and from the most obvious errors in implementing them to the subtlest ones—some of which also happen to be the most gruesome.

In brief, smart pointers are C++ objects that simulate simple pointers by implementing `operator->` and the unary `operator*`. In addition to sporting pointer syntax and semantics, smart pointers often perform useful tasks—such as memory management or locking—under the covers, thus freeing the application from carefully managing the lifetime of pointed-to objects.

This chapter not only discusses smart pointers but also implements a `SmartPtr` class template. `SmartPtr` is designed around policies (see Chapter 1), and the result is a smart pointer that has the exact levels of safety, efficiency, and ease of use that you want.

After reading this chapter, you will be an expert in smart pointer issues such as the following:

- The advantages and disadvantages of smart pointers
- Ownership management strategies
- Implicit conversions
- Tests and comparisons
- Multithreading issues

This chapter implements a generic `SmartPtr` class template. Each section presents one implementation issue in isolation. At the end, the implementation puts all the pieces together. In addition to understanding the design rationale of `SmartPtr`, you will know how to use, tweak, and extend it.

7.1 Smart Pointers 101

So what's a smart pointer? A smart pointer is a C++ class that mimics a regular pointer in syntax and some semantics, but it does more. Because smart pointers to different type of objects tend to have a lot of code in common, almost all good-quality smart pointers in existence are templated by the pointee type, as you can see in the following code:

```
template <class T>
class SmartPtr
{
public:
   explicit SmartPtr(T* pointee) : pointee_(pointee);
   SmartPtr& operator=(const SmartPtr& other);
   ~SmartPtr();
   T& operator*() const
   {
      ...
      return *pointee_;
   }
   T* operator->() const
   {
      ...
      return pointee_;
   }
private:
   T* pointee_;
   ...
};
```

`SmartPtr<T>` aggregates a pointer to `T` in its member variable `pointee_`. That's what most smart pointers do. In some cases, a smart pointer might aggregate some handles to data and compute the pointer on the fly.

The two operators give `SmartPtrToSomething` pointer-like syntax and semantics. That is, you can write

```
class Widget
{
public:
   void Fun();
};

SmartPtr<Widget> sp(new Something);
sp->Fun();
(*sp).Fun();
```

Aside from the definition of `sp`, nothing reveals it as not being a pointer. This is the mantra of smart pointers: You can replace pointer definitions with smart pointer definitions without incurring major changes to your application's code. You thus get extra goodies with ease. Minimizing code changes is very appealing and vital for getting large applications to use smart pointers. As you will soon see, however, smart pointers are not a free lunch.

7.2 The Deal

But what's the deal with smart pointers? you might ask. What do you gain by replacing simple pointers with smart pointers? The explanation is simple. Smart pointers have value semantics, whereas some simple pointers do not.

An object with value semantics is an object that you can *copy* and *assign to.* Type `int` is the perfect example of a first-class object. You can create, copy, and change integer values freely. A pointer that you use to iterate in a buffer also has value semantics—you initialize it to point to the beginning of the buffer, and you bump it until you reach the end. Along the way, you can copy its value to other variables to hold temporary results.

With pointers that hold values allocated with `new`, however, the story is very different. Once you have written

```
Widget* p = new Widget;
```

the variable `p` not only points to, but also *owns,* the memory allocated for the `Widget` object. This is because later you must issue `delete p` to ensure that the `Widget` object is destroyed and its memory is released. If in the line after the line just shown you write

```
p = 0; // assign something else to p
```

you lose ownership of the object previously pointed to by `p`, and you have no chance at all to get a grip on it again. You have a resource leak, and resource leaks never help.

Furthermore, when you copy `p` into another variable, the compiler does not automatically manage the ownership of the memory to which the pointer points. All you get is two raw pointers pointing to the same object, and you have to track them even more carefully because double deletions are even more catastrophic than no deletion. Consequently, pointers to allocated objects do *not* have value semantics—you cannot copy and assign to them at your will.

Smart pointers can be of great help in this area. Most smart pointers offer *ownership management* in addition to pointer-like behavior. Smart pointers can figure out how ownership evolves, and their destructors can release the memory according to a well-defined strategy. Many smart pointers hold enough information to take full initiative in releasing the pointed-to object.

Smart pointers may manage ownership in various ways, each appropriate to a category of problems. Some smart pointers transfer ownership automatically: After you copy a smart pointer to an object, the source smart pointer becomes null, and the destination points to (and holds ownership of) the object. This is the behavior implemented by the standard-provided `std::auto_ptr`. Other smart pointers implement reference counting: They track the total count of smart pointers that point to the same object, and when this count goes down to zero, they `delete` the pointed-to object. Finally, some others duplicate their pointed-to object whenever you copy them.

In short, in the smart pointers' world, ownership is an important topic. By providing ownership management, smart pointers are able to support integrity guarantees and full value semantics. Because ownership has much to do with constructing, copying, and destroying smart pointers, it's easy to figure out that these are the most vital functions of a smart pointer.

The following few sections discuss various aspects of smart pointer design and implementation. The goal is to render smart pointers as compatible with raw pointers as possible, but not closer. It's a contradictory goal: After all, if your smart pointers behave *exactly* like dumb pointers, they *are* dumb pointers.

In implementing compatibility between smart pointers and raw pointers, there is a thin line between nicely filling compatibility checklists and paving the way toward chaos. You will find that adding seemingly worthwhile features might expose the clients to costly risks. Much of the craft of implementing good smart pointers consists of carefully balancing their set of features.

7.3 Smart Pointers' Storage

To start, let's ask a fundamental question about smart pointers. Is `pointee_`'s type necessarily `T*`? If not, what else could it be? In generic programming, you should always ask yourself questions like these. Each type that's hardcoded in a piece of generic code decreases the genericity of the code. Hardcoded types are to generic code what magic constants are to regular code.

In several situations, it is worthwhile to allow customizing the pointee type. One situation is when you deal with nonstandard pointer modifiers. In the 16-bit Intel 80x86 days, you could qualify pointers with modifiers like `__near`, `__far`, and `__huge`. Other segmented memory architectures use similar modifiers.

Another situation is when you want to layer smart pointers. What if you have a `LegacySmartPtr<T>` smart pointer implemented by someone else, and you want to enhance it? Would you derive from it? That's a risky decision. It's better to wrap the legacy smart pointer into a smart pointer of your own. This is possible because the inner smart pointer supports pointer syntax. From the outer smart pointer's viewpoint, the pointee type is not `T*` but `LegacySmartPtr<T>`.

There are interesting applications of smart pointer layering, mainly because of the mechanics of `operator->`. When you apply `operator->` to a type that's not a built-in pointer, the compiler does an interesting thing. After looking up and applying the user-defined `operator->` to that type, it applies `operator->` again to the result. The compiler keeps doing this recursively until it reaches a pointer to a built-in type, and only then proceeds with member access. It follows that a smart pointer's `operator->` does not have to return a pointer. It can return an object that in turn implements `operator->`, without changing the use syntax.

This leads to a very interesting idiom: pre- and postfunction calls (Stroustrup 2000). If you return an object of type `PointerType` by value from `operator->`, the sequence of execution is as follows:

1. Constructor of `PointerType`
2. `PointerType::operator->` called; likely returns a pointer to an object of type `PointeeType`
3. Member access for `PointeeType`—likely a function call
4. Destructor of `PointerType`

In a nutshell, you have a nifty way of implementing locked function calls. This idiom has broad uses with multithreading and locked resource access. You can have `PointerType`'s constructor lock the resource, and then you can access the resource; finally, `Pointer Type`'s destructor unlocks the resource.

The generalization doesn't stop here. The syntax-oriented "pointer" part might sometimes become dim in comparison with the powerful resource management techniques that are included in smart pointers. It follows that, in rare cases, smart pointers could drop the pointer syntax. An object that does not define `operator->` and `operator*` violates the definition of a smart pointer, but there are objects that do deserve smart pointer–like treatment, although they are not, strictly speaking, smart pointers.

Look at real-world APIs and applications. Many operating systems foster *handles* as accessors to certain internal resources, such as windows, mutexes, or devices. Handles are intentionally obfuscated pointers; one of their purposes is to prevent their users from manipulating critical operating system resources directly. Most of the time, handles are integral values that are indices in a hidden table of pointers. The table provides the additional level of indirection that protects the inner system from the application programmers. Although they don't provide an `operator->`, handles resemble pointers in semantics and in the way they are managed.

For such a smart resource, it does not make sense to provide `operator->` or `operator*`. However, you do take advantage of all the resource management techniques that are specific to smart pointers.

To generalize the type universe of smart pointers, we distinguish three potentially distinct types in a smart pointer:

- *The storage type.* This is the type of `pointee_`. By "default"—in regular smart pointers—it is a raw pointer.
- *The pointer type.* This is the type returned by `operator->`. It can be different from the storage type if you want to return a proxy object instead of just a pointer. (You will find an example of using proxy objects later in this chapter.)
- *The reference type.* This is the type returned by `operator*`.

It would be useful if `SmartPtr` supported this generalization in a flexible way. Thus, the three types mentioned here ought to be abstracted in a policy called Storage.

In conclusion, smart pointers can, and should, generalize their pointee type. To do this, `SmartPtr` abstracts three types in a Storage policy: the stored type, the pointer type, and the reference type. Not all types necessarily make sense for a given `SmartPtr` instantiation. Therefore, in rare cases (handles), a policy might disable access to `operator->` or `operator*` or both.

7.4 Smart Pointer Member Functions

Many existing smart pointer implementations allow operations through member functions, such as `Get` for accessing the pointee object, `Set` for changing it, and `Release` for taking over ownership. This is the obvious and natural way of encapsulating `SmartPtr`'s functionality.

However, experience has proven that member functions are not very suitable for smart pointers. The reason is that the interaction between member function calls for the smart pointer for the *pointed-to* object can be extremely confusing.

Suppose, for instance, that you have a `Printer` class with member functions such as `Acquire` and `Release`. With `Acquire` you take ownership of the printer so that no other application prints to it, and with `Release` you relinquish ownership. As you use a smart pointer to Printer, you may notice a strange syntactical closeness to things that are very far apart semantically.

```
SmartPtr<Printer> spRes = ...;
spRes->Acquire(); // acquire the printer
... print a document ...
spRes->Release(); // release the printer
spRes.Release();  // release the pointer to the printer
```

The user of SmartPtr now has access to two totally different worlds: the world of the pointed-to object members and the world of the smart pointer members. A matter of a dot or an arrow thinly separates the two worlds.

On the face of it, C++ does force you routinely to observe certain slight differences in syntax. A Pascal programmer learning C++ might even feel that the slight syntactic difference between & and && is an abomination. Yet C++ programmers don't even blink at it. They are trained by habit to distinguish such syntax matters easily.

However, smart pointer member functions defeat training by habit. Raw pointers don't have member functions, so C++ programmers' eyes are not habituated to detect and distinguish dot calls from arrow calls. The compiler does a good job at that: If you use a dot after a raw pointer, the compiler will yield an error. Therefore, it is easy to imagine, and experience proves, that even seasoned C++ programmers find it extremely disturbing that both sp.Release() and sp->Release() compile flag-free but do very different things. The cure is simple: A smart pointer should not use member functions. SmartPtr uses only nonmember functions. These functions become friends of the smart pointer class.

Overloaded functions can be just as confusing as member functions of smart pointers, but there is an important difference. C++ programmers already use overloaded functions. Overloading is an important part of the C++ language and is used routinely in library and application development. This means that C++ programmers *do* pay attention to differences in function call syntax—such as Release(*sp) versus Release(sp)—in writing and reviewing code.

The only functions that necessarily remain members of SmartPtr are the constructors, the destructor, operator=, operator->, and unary operator*. All other operations of SmartPtr are provided through named nonmember functions.

For reasons of clarity, SmartPtr does not have any named member functions. The only functions that access the pointee object are GetImpl, GetImplRef, Reset, and Release, which are defined at namespace level.

```
template <class T> T* GetImpl(SmartPtr<T>& sp);
template <class T> T*& GetImplRef(SmartPtr<T>& sp);
template <class T> void Reset(SmartPtr<T>& sp, T* source);
template <class T> void Release(SmartPtr<T>& sp, T*& destination);
```

- GetImpl returns the pointer object stored by SmartPtr.
- GetImplRef returns *a reference* to the pointer object stored by SmartPtr. GetImplRef allows you to change the underlying pointer, so it requires extreme care in use.
- Reset resets the underlying pointer to another value, releasing the previous one.
- Release releases ownership of the smart pointer, giving its user the responsibility of managing the pointee object's lifetime.

The actual declarations of these four functions in Loki are slightly more elaborate. They don't assume that the type of the pointer object stored by `SmartPtr` is `T*`. As discussed in Section 7.3, the Storage policy defines the pointer type. Most of the time, it's a straight pointer, except for exotic implementations of Storage, when it might be a handle or an elaborate type.

7.5 Ownership-Handling Strategies

Ownership handling is often the most important *raison d'être* of smart pointers. Usually, from their clients' viewpoint, smart pointers own the objects to which they point. A smart pointer is a first-class value that takes care of deleting the pointed-to object under the covers. The client can intervene in the pointee object's lifetime by issuing calls to helper management functions.

For implementing self-ownership, smart pointers must carefully track the pointee object, especially during copying, assignment, and destruction. Such tracking brings some overhead in space, time, or both. An application should settle on the strategy that best fits the problem at hand and does not cost too much.

The following subsections discuss the most popular ownership management strategies and how `SmartPtr` implements them.

7.5.1 Deep Copy

The simplest strategy applicable is to copy the pointee object whenever you copy the smart pointer. If you ensure this, there is only one smart pointer for each pointee object. Therefore, the smart pointer's destructor can safely `delete` the pointee object. Figure 7.1 depicts the state of affairs if you use smart pointers with deep copy.

At first glance, the deep copy strategy sounds rather dull. It seems like the smart pointer does not add any value over regular C++ value semantics. Why would you make the effort of using a smart pointer, when simple pass by value of the pointee object works just as well?

The answer is *support for polymorphism.* Smart pointers are vehicles for transporting polymorphic objects safely. You hold a smart pointer to a base class, which might actually point to a derived class. When you copy the smart pointer, you want to copy its polymorphic behavior, too. It's interesting that you don't exactly know what behavior and state you are dealing with, but you certainly need to duplicate that behavior and state.

Because deep copy most often deals with polymorphic objects, the following naive implementation of the copy constructor is wrong:

```
template <class T>
class SmartPtr
{
public:
   SmartPtr(const SmartPtr& other)
   : pointee_(new T(*other.pointee_))
   {
   }
   ...
};
```

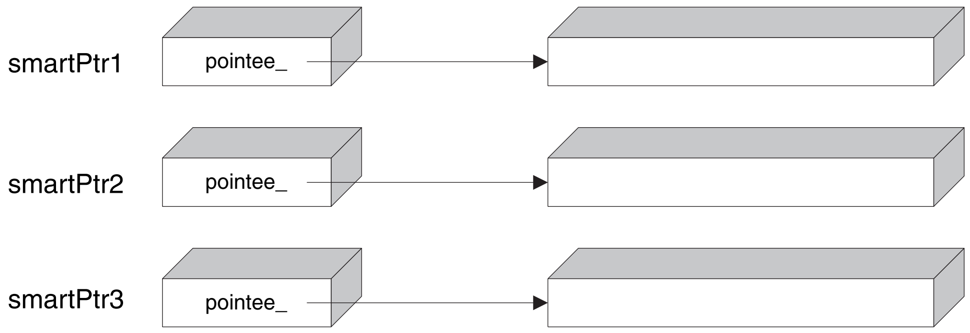

Figure 7.1: Memory layout for smart pointers with eager copy

Say you copy an object of type `SmartPtr<Widget>`. If `other` points to an instance of a class `ExtendedWidget` that derives from `Widget`, the copy constructor above copies only the `Widget` part of the `ExtendedWidget` object. This phenomenon is known as *slicing*—only the `Widget` "slice" of the object of the presumably larger type `ExtendedWidget` gets copied. Slicing is most often undesirable. It is a pity that C++ allows slicing so easily—a simple call by value slices objects without any warning.

Chapter 8 discusses cloning in depth. As shown there, the classic way of obtaining a polymorphic clone for a hierarchy is to define a virtual `Clone` function and implement it as follows:

```
class AbstractBase
{
   ...
   virtual Base* Clone() = 0;
};

class Concrete : public AbstractBase
{
   ...
   virtual Base* Clone()
   {
      return new Concrete(*this);
   }
};
```

The `Clone` implementation must follow the same pattern in all derived classes; in spite of its repetitive structure, there is no reasonable way to automate defining the `Clone` member function (beyond macros, that is).

A generic smart pointer cannot count on knowing the exact name of the cloning member function—maybe it's `clone`, or maybe `MakeCopy`. Therefore, the most flexible approach is to parameterize `SmartPtr` with a policy that addresses cloning.

7.5.2 Copy on Write

Copy on write (COW, as fondly called by its fans) is an optimization technique that avoids unnecessary object copying. The idea that underlies COW is to clone the pointee object at the first attempt of modification; until then, several pointers can share the same object.

Smart pointers, however, are not the best place to implement COW, because smart pointers cannot differentiate between calls to `const` and non-`const` member functions of the pointee object. Here is an example:

```
template <class T>
class SmartPtr
{
public:
   T* operator->() { return pointee_; }
   ...
};

class Foo
{
public:
   void ConstFun() const;
   void NonConstFun();
};

...
SmartPtr<Foo> sp;
sp->ConstFun(); // invokes operator->, then ConstFun
sp->NonConstFun(); // invokes operator->, then NonConstFun
```

The same `operator->` gets invoked for both functions called; therefore, the smart pointer does not have any clue whether to make the COW or not. Function invocations for the pointee object happen somewhere beyond the reach of the smart pointer. (Section 7.11 explains how `const` interacts with smart pointers and the objects they point to.)

In conclusion, COW is effective mostly as an implementation optimization for full-featured classes. Smart pointers are at too low a level to implement COW semantics effectively. Of course, smart pointers can be a good building block in implementing COW for a class.

The `SmartPtr` implementation in this chapter does not provide support for COW.

7.5.3 Reference Counting

Reference counting is the most popular ownership strategy used with smart pointers. Reference counting tracks the number of smart pointers that point to the same object. When that number goes to zero, the pointee object is deleted. This strategy works very well if you don't break certain rules—for instance, you should not keep dumb pointers and smart pointers to the same object.

The actual counter must be shared among smart pointer objects, leading to the structure

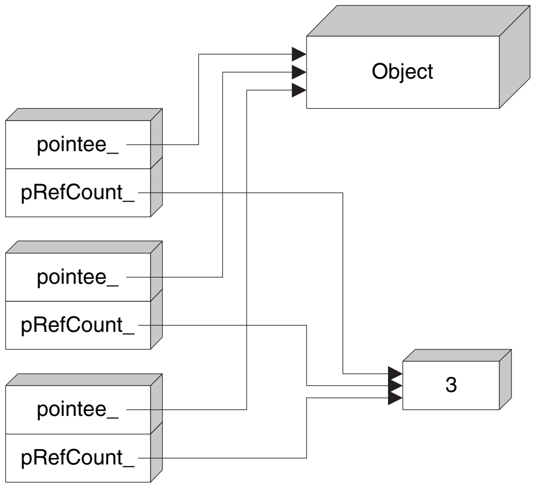

Figure 7.2: Three reference-counted smart pointers pointing to the same object

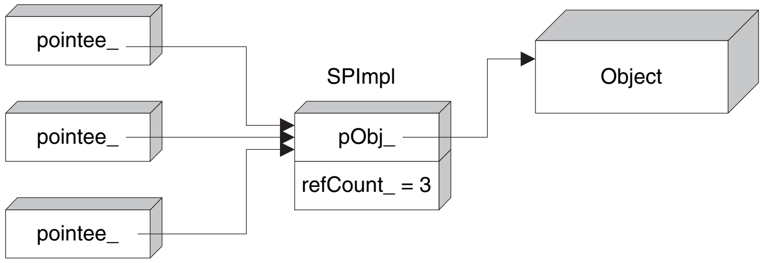

Figure 7.3: An alternate structure of reference-counted pointers

depicted in Figure 7.2. Each smart pointer holds a pointer to the reference counter (`pRefCount_` in Figure 7.2) in addition to the pointer to the object itself. This usually doubles the size of the smart pointer, which may or may not be an acceptable overhead, depending on your needs and constraints.

There is another, subtler, overhead issue. Reference-counted smart pointers must store the reference counter on the free store. The problem is that in many implementations, the default C++ free store allocator is remarkably slow and wasteful of space when it comes to allocating small objects, as discussed in Chapter 4. (Obviously, the reference count, typically occupying 4 bytes, does qualify as a small object.) The overhead in speed stems from slow algorithms in finding available chunks of memory, and the overhead in size is incurred by the bookkeeping information that the allocator holds for each chunk.

The relative size overhead can be partially mitigated by holding the pointer and the reference count together, as in Figure 7.3. The structure in Figure 7.3 reduces the size of the smart pointer back to that of a pointer, but at the expense of access speed: The pointee ob-

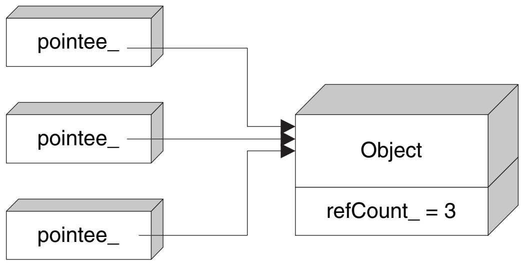

Figure 7.4: Intrusive reference counting

ject is an extra indirection level away. This is a considerable drawback because you typically use a smart pointer several times, whereas you obviously construct and destroy it only once.

The most efficient solution is to hold the reference counter in the pointee object itself, as shown in Figure 7.4. This way `SmartPtr` is just the size of a pointer, and there is no extra overhead at all. This technique is known as *intrusive reference counting*, because the reference count is an "intruder" in the pointee—it semantically belongs to the smart pointer. The name also gives a hint about the Achilles' heel of the technique: You must design up front or modify the pointee class to support reference counting.

A generic smart pointer should use intrusive reference counting where available and implement a nonintrusive reference counting scheme as an acceptable alternative. For implementing nonintrusive reference counting, the small-object allocator presented in Chapter 4 can help a great deal. The `SmartPtr` implementation using nonintrusive reference counting leverages the small-object allocator, thus slashing the performance overhead caused by the reference counter.

7.5.4 Reference Linking

Reference linking relies on the observation that you don't really need the actual count of smart pointer objects pointing to one pointee object; you only need to detect when that count goes down to zero. This leads to the idea of keeping an "ownership list," as shown in Figure 7.5.[1]

All `SmartPtr` objects that point to a given pointee form a doubly linked list. When you create a new `SmartPtr` from an existing `SmartPtr`, the new object is appended to the list; `SmartPtr`'s destructor takes care of removing the destroyed object from the list. When the list becomes empty, the pointee object is deleted.

The doubly linked list structure fits reference tracking like a glove. You cannot use a singly linked list because removals from such a list take linear time. You cannot use a vector because the `SmartPtr` objects are not contiguous (and removals from vectors take linear time anyway). You need a structure sporting constant-time append, constant-time remove, and constant-time empty detection. This bill is fit precisely and exclusively by doubly linked lists.

[1]Risto Lankinen described the reference-linking mechanism on Usenet in November 1995.

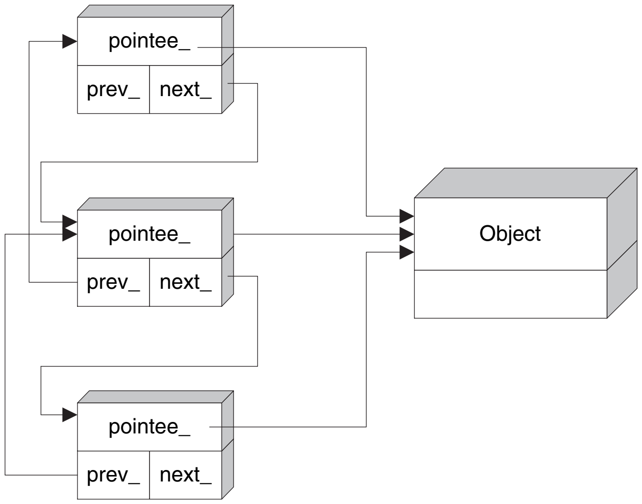

Figure 7.5: Reference linking in action

In a reference-linking implementation, each `SmartPtr` object holds two extra pointers—one to the next element and one to the previous element.

The advantage of reference linking over reference counting is that the former does not use extra free store, which makes it more reliable: Creating a reference-linked smart pointer cannot fail. The disadvantage is that reference linking needs more memory for its bookkeeping (three pointers versus only one pointer plus one integer). Also, reference counting should be a bit speedier—when you copy smart pointers, only an indirection and an increment are needed. The list management is slightly more elaborate. In conclusion, you should use reference linking only when the free store is scarce. Otherwise, prefer reference counting.

To wrap up the discussion on reference count management strategies, let's note a significant disadvantage that they have. Reference management—be it counting or linking—is victim to the resource leak known as *cyclic reference*. Imagine an object `A` holds a smart pointer to an object `B`. Also, object `B` holds a smart pointer to `A`. These two objects form a cyclic reference; even though you don't use any of them anymore, they use each other. The reference management strategy cannot detect such cyclic references, and the two objects remain allocated forever. The cycles can span multiple objects, closing circles that often reveal unexpected—and very hard to debug—dependencies.

In spite of this, reference management is a robust, speedy ownership-handling strategy. If used with precaution, reference management makes application development significantly easier.

7.5.5 Destructive Copy

Destructive copy does exactly what you think it does: During copying, it destroys the object being copied. In the case of smart pointers, destructive copy destroys the source smart pointer by taking its pointee object and passing it to the destination smart pointer. The std::auto_ptr class template features destructive copy.

In addition to being suggestive about the action taken, "destructive" also vividly describes the dangers associated with this strategy. Misusing destructive copy may have destructive effects on your program data, your program correctness, and your brain cells.

Smart pointers may use destructive copy to ensure that at any time there is only one smart pointer pointing to a given object. During the copying or assignment of a smart pointer to another, the "living" pointer is passed to the destination of the copy, and source's pointee_ becomes zero. The following code illustrates a copy constructor and an assignment operator of a simple SmartPtr featuring destructive copy.

```
template <class T>
class SmartPtr
{
public:
   SmartPtr(SmartPtr& src)
   {
      pointee_ = src.pointee_;
      src.pointee_ = 0;
   }
   SmartPtr& operator=(SmartPtr& src)
   {
      if (this != &src)
      {
         delete pointee_;
         pointee_ = src.pointee_;
         src.pointee_ = 0;
      }
      return *this;
   }
   ...
};
```

C++ etiquette calls for the right-hand side of the copy constructor and the assignment operator to be a reference to a const object. Classes that foster destructive copy break this convention for obvious reasons. Because etiquette exists for a reason, you should expect negative consequences if you break it. Indeed, here it is:

```
void Display(SmartPtr<Something> sp);
...
SmartPtr<Something> sp(new Something);
Display(sp);  // sinks sp
```

Although Display means no harm to its argument (accepts it by value), it acts like a maelstrom of smart pointers: It sinks any smart pointer passed to it. After calling Display(sp), sp holds the null pointer.

Because they do not support value semantics, smart pointers with destructive copy cannot be stored in containers and in general must be handled with almost as much care as raw pointers.

The ability to store smart pointers in a container is very important. Containers of raw pointers make manual ownership management tricky, so many containers of pointers can use smart pointers to good advantage. Smart pointers with destructive copy, however, do not mix with containers.

On the bright side, smart pointers with destructive copy have significant advantages:

- They incur almost no overhead.
- They are good at enforcing ownership transfer semantics. In this case, you use the "maelstrom effect" described earlier to your advantage: You make it clear that your function takes over the passed-in pointer.
- They are good as return values from functions. If the smart pointer implementation uses a certain trick,[2] you can return smart pointers with destructive copy from functions. This way, you can be sure that the pointee object gets destroyed if the caller doesn't use the return value.
- They are excellent as stack variables in functions with multiple return paths. You don't have to remember to delete the pointee object manually—the smart pointer takes care of this for you.

The destructive copy strategy is used by the standard-provided `std::auto_ptr`. This brings destructive copy another important advantage:

- Smart pointers with destructive copy semantics are the only smart pointers that the standard provides, which means that many programmers will get used to their behavior sooner or later.

For these reasons, the `SmartPtr` implementation should provide optional support for destructive copy semantics.

Smart pointers use various ownership semantics, each having different trade-offs. The most important techniques are deep copy, reference counting, reference linking, and destructive copy. `SmartPtr` implements all these strategies through an Ownership policy, allowing its users to choose the one that best fits an application's needs. The default strategy is reference counting.

7.6 The Address-of Operator

In striving to make smart pointers as indistinguishable as possible from their native counterparts, designers stumbled upon an obscure operator that is on the list of overloadable operators: unary `operator&`, the *address-of operator.*[3]

An implementer of smart pointers might choose to overload the address-of operator like this:

```
template <class T>
class SmartPtr
{
public:
```

[2]Invented by Greg Colvin and Bill Gibbons for `std::auto_ptr`.
[3]*Unary* `operator&` is to differentiate it from binary `operator&`, which is the bitwise AND operator.

```
      T** operator&()
      {
         return &pointee_;
      }
      ...
   };
```

After all, if a smart pointer is to simulate a pointer, then its address must be substitutable for the address of a regular pointer. This overload makes code like the following possible:

```
void Fun(Widget** pWidget);
...
SmartPtr<Widget> spWidget(...);
Fun(&spWidget); // okay, invokes operator* and obtains a
            // pointer to pointer to Widget
```

It seems very desirable to have such an accurate compatibility between smart pointers and dumb pointers, but overloading the unary `operator&` is one of those clever tricks that can do more harm than good. There are two reasons why overloading unary `operator&` is not too good an idea.

One reason is that exposing the address of the pointed-to object implies giving up any automatic ownership management. When a client freely accesses the address of the raw pointer, any helper structures that the smart pointer holds, such as reference counts, become invalid for all purposes. While the client deals directly with the address of the raw pointer, the smart pointer is completely unconscious.

The second reason, a more pragmatic one, is that overloading unary `operator&` makes the smart pointer unusable with STL containers. Actually, overloading unary `operator&` for a type pretty much makes generic programming impossible for that type, because the address of an object is too fundamental a property to play with naively. Most generic code assumes that applying `&` to an object of type `T` returns an object of type `T*`—you see, address-of is a fundamental concept. If you defy this concept, generic code behaves strangely either at compile time or—worse—at runtime.

Thus, it is not recommended to overload unary `operator&` for smart pointers, and for any objects in general. `SmartPtr` does not overload unary `operator&`.

7.7 Implicit Conversion to Raw Pointer Type

Consider this code:

```
void Fun(Something* p);
...
SmartPtr<Something> sp(new Something);
Fun(sp); // OK or error?
```

Should this code compile or not? Following the “maximum compatibility” line of thought, the answer is yes.

Technically, it is very simple to render the previous code compilable by introducing a user-defined conversion, like so:

```
template <class T>
class SmartPtr
{
public:
   operator T*() // user-defined conversion to T*
   {
      return pointee_;
   }
   ...
};
```

However, this is not the end of the story.

User-defined conversions in C++ have an interesting history. Back in the 1980s, when user-defined conversions were introduced, most programmers considered them a great invention. User-defined conversions promised a more unified type system, expressive semantics, and the ability to define new types that were indistinguishable from built-in ones. With time, however, user-defined conversions revealed themselves as awkward and potentially dangerous. They might become dangerous especially when they expose handles to internal data (Meyers 1998a, Item 29), which is precisely the case with the `operator T*` in the previous code. That's why you should think carefully before allowing automatic conversions for the smart pointers you design.

One potential danger comes inherently from giving the user unattended access to the raw pointer that the smart pointer wraps. Passing the raw pointer around defeats the inner workings of the smart pointer. Once unleashed from the confines of its wrapper, the raw pointer can easily become a threat to program sanity again, just as it was before introducing any smart pointers at all.

Another danger is that user-defined conversions pop up unexpectedly, even when you don't need them. Consider the following code:

```
SmartPtr<Something> sp;
...
// A gross semantic error
// However, it goes undetected at compile time
delete sp;
```

The compiler matches operator `delete` with the user-defined conversion to `T*`. At runtime, `operator T*` is called, and `delete` is applied to the result of it. This is certainly not what you want to do to a smart pointer, because it is supposed to manage ownership itself. An extra unwitting `delete` call throws out the window all the careful ownership management that the smart pointer performs under the covers.

There are quite a few ways to prevent the `delete` call from compiling. Some of them are very ingenious (Meyers 1996). One that's very effective and easy to implement is to make the call to `delete` intentionally *ambiguous.* You can achieve this by providing *two* automatic conversions to types that are susceptible to a call to `delete`. One type is `T*` itself, and the other can be `void*`.

```
template <class T>
class SmartPtr
{
public:
   operator T*() // User-defined conversion to T*
   {
      return pointee_;
   }
   operator void*() // Added–conversion to void*
   {
      return pointee_;
   }
   ...
};
```

A call to `delete` against such a smart pointer object is ambiguous. The compiler cannot decide which conversion to apply, and the trick above exploits this indecision to good advantage.

Don't forget that disabling the `delete` operator was only a part of the issue. Whether to provide an automatic conversion to a raw pointer remains an important decision in implementing a smart pointer. It's too dangerous just to let it in, yet too convenient to rule it out. The final `SmartPtr` implementation will give you a choice about that.

However, forbidding implicit conversion does not necessarily mean that any access to the raw pointer is gone; it is often necessary to gain access to the raw pointer. Therefore, all smart pointers do provide *explicit* access to their wrapped pointer via a call to a function, like this:

```
void Fun(Something* p);
...
SmartPtr<Something> sp;
Fun(GetImpl(sp)); // OK, explicit conversion always allowed
```

It's not whether you can get to the wrapped pointer or not; it's how easy it is. This may seem like a minor difference, but it's actually very important. An implicit conversion happens without the programmer or the maintainer noticing or even knowing it. An explicit conversion—as is the call to `GetImpl`—passes through the mind, the understanding, and the fingers of the programmer and remains written there for everybody to see it.

Implicit conversion from the smart pointer type to the raw pointer type is desirable, but sometimes dangerous. `SmartPtr` provides this implicit conversion as a choice. The default is on the safe side—no implicit conversions. Explicit access is always available through the `GetImpl` function.

7.8 Equality and Inequality

C++ teaches its users that any clever trick like the one presented in the previous section (intentional ambiguity) establishes a new context, which in turn may have unexpected ripples.

Consider tests for equality and inequality of smart pointers. A smart pointer should support the full syntax that raw pointers support. Let's first focus on comparisons against zero. Programmers expect the following tests to compile and run as they do for a raw pointer.

```
SmartPtr<Something> sp1, sp2;
Something* p;
...
if (sp1)    // Test 1: direct test for non-null pointer
   ...
if (!sp1)      // Test 2: direct test for null pointer
   ...
if (sp1 == 0)   // Test 3: explicit test for null pointer
   ...
if (sp1 == sp2) // Test 4: comparison of two smart pointers
   ...
if (sp1 == p)   // Test 5: comparison with a raw pointer
   ...
```

There are more tests than depicted here if you consider symmetry and `operator!=`. If we solve the equality tests, we can easily define the corresponding symmetric and inequality tests.

There is an unfortunate interference between the solution to the previous issue (prevent `delete` from compiling) and a possible solution to this issue. With one user-defined conversion to the pointee type, most of the test expressions (except test 4) compile successfully and run as expected. The downside is that you can accidentally call the `delete` operator against the smart pointer. With two user-defined conversions (intentional ambiguity), you detect wrongful `delete` calls, but none of these tests compiles anymore—they have become ambiguous too.

An additional user-defined conversion to `bool` helps, but this, to nobody's surprise, introduces new trouble. Given this smart pointer:

```
template <class T>
class SmartPtr
{
public:
   operator bool() const
   {
      return pointee_ != 0;
   }
   ...
};
```

the four tests compile, but so do the following nonsensical operations:

```
SmartPtr<Apple> sp1;
SmartPtr<Orange> sp2; // Orange is unrelated to Apple
if (sp1 == sp2)   // Converts both pointers to bool
                 // and compares results
   ...
if (sp1 != sp2)   // Ditto
   ...
```

```
bool b = sp1;          // The conversion allows this, too
if (sp1 * 5 == 200)   // Ouch! SmartPtr behaves like an integral
                 // type!
   ...
```

As you can see, it's either not at all or too much: Once you add a user-defined conversion to `bool`, you allow `SmartPtr` to act as a `bool` in many more situations than you actually wanted. For all practical purposes, defining an `operator bool` for a smart pointer is not a smart solution.

A true, complete, rock-solid solution to this dilemma is to go all the way and overload each and every operator separately. This way any operation that makes sense for the bare pointer makes sense for the smart pointer, and nothing else. Here is the code that implements this idea.

```
template <class T>
class SmartPtr
{
public:
   bool operator!() const // Enables "if (!sp) ..."
   {
      return pointee_ == 0;
   }
   inline friend bool operator==(const SmartPtr& lhs,
      const T* rhs)
   {
      return lhs.pointee_ == rhs;
   }
   inline friend bool operator==(const T* lhs,
      const SmartPtr& rhs)
   {
      return lhs == rhs.pointee_;
   }
   inline friend bool operator!=(const SmartPtr& lhs,
      const T* rhs)
   {
      return lhs.pointee_ != rhs;
   }
   inline friend bool operator!=(const T* lhs,
      const SmartPtr& rhs)
   {
      return lhs != rhs.pointee_;
   }
   ...
};
```

Yes, it's a pain, but this approach solves the problems with almost all comparisons, including the tests against the literal zero. What the forwarding operators in this code do is to pass operators that client code applies to the smart pointer on to the raw pointer that the smart pointer wraps. No simulation can be more realistic than that.

We still didn't solve the problem completely. If you provide an automatic conversion to the pointee type, there still is risk of ambiguities. Suppose you have a class Base and a class Derived that inherits Base. Then the following code makes practical sense yet is ill formed due to ambiguity.

```
SmartPtr<Base> sp;
Derived* p;
...
if (sp == p) {} // error! Ambiguity between:
            // '(Base*)sp == (Base*)p'
            // and 'operator==(sp, (Base*)p)'
```

Indeed, smart pointer development is not for the faint of heart.

We're not out of bullets, though. In addition to the definitions of operator== and operator!=, we can add *templated* versions of them, as you can see in the following code:

```
template <class T>
class SmartPtr
{
public:
   ... as above ...
   template <class U>
   inline friend bool operator==(const SmartPtr& lhs,
      const U* rhs)
   {
      return lhs.pointee_ == rhs;
   }
   template <class U>
   inline friend bool operator==(const U* lhs,
      const SmartPtr& rhs)
   {
      return lhs == rhs.pointee_;
   }
   ... similarly defined operator!= ...
};
```

The templated operators are "greedy" in the sense that they match comparisons with any pointer type whatsoever, thus eating the ambiguity.

If that's the case, why should we keep the nontemplated operators—the ones that take the pointee type? They never get a chance to match, because the template matches any pointer type, including the pointee type itself.

The rule that "never" actually means "almost never" applies here, too. In the test if (sp == 0), the compiler tries the following matches.

- *The templated operators.* They don't match because zero is not a pointer type. A literal zero can be implicitly converted to a pointer type, but template matching does not include conversions.
- *The nontemplated operators.* After eliminating the templated operators, the compiler tries the nontemplated ones. One of these operators kicks in through an implicit conversion from the literal zero to the pointee type. Had the nontemplated operators not existed, the test would have been an error.

In conclusion, we need *both* the nontemplated and the templated comparison operators.

Let's see now what happens if we compare two SmartPtrs instantiated with different types.

```
SmartPtr<Apple> sp1;
SmartPtr<Orange> sp2;
if (sp1 == sp2)
   ...
```

The compiler chokes on the comparison because of an ambiguity: Each of the two SmartPtr instantiations defines an operator==, and the compiler does not know which one to choose. We can dodge this problem by defining an "ambiguity buster" as shown:

```
template <class T>
class SmartPtr
{
public:
   // Ambiguity buster
   template <class U>
   bool operator==(const SmartPtr<U>& rhs) const
   {
      return pointee_ == rhs.pointee_;
   }
   // Similarly for operator!=
   ...
};
```

This newly added operator is a member that specializes exclusively in comparing SmartPtr<...> objects. The beauty of this ambiguity buster is that it makes smart pointer comparisons act like raw pointer comparisons. If you compare two smart pointers to Apple and Orange, the code will be essentially equivalent to comparing two raw pointers to Apple and Orange. If the comparison makes sense, then the code compiles; otherwise, it's a compile-time error.

```
SmartPtr<Apple> sp1;
SmartPtr<Orange> sp2;
if (sp1 == sp2)   // Semantically equivalent to
              // sp1.pointee_ == sp2.pointee_
   ...
```

There is one unsatisfied syntactic artifact left, namely, the direct test if (sp). Here life becomes really interesting. The if statement applies only to expressions of arithmetic and pointer type. Consequently, to allow if (sp) to compile, we must define an automatic conversion to either an arithmetic or a pointer type.

A conversion to arithmetic type is not recommended, as the experience with operator bool earlier witnesses. A pointer is not an arithmetic type, period. A conversion to a pointer type makes a lot more sense, and here the problem branches.

If you want to provide automatic conversions to the pointee type (see previous section), then you have two choices: You either risk unattended calls to operator delete, or you forgo the `if (sp)` test. The tiebreaker is between the lack of a convenience and a risky life. The winner is safety, so you cannot write `if (sp)`. Instead, you can choose between `if(sp != 0)` and the more baroque `if(!!sp)`. End of story.

If you don't want to provide automatic conversions to the pointee type, there is an interesting trick you can do to make `if (sp)` possible. Inside the `SmartPtr` class template, define an inner class `Tester` and define a conversion to `Tester*`, as shown in the following code:

```
template <class T>
class SmartPtr
{
   class Tester
   {
      void operator delete(void*);
   };
public:
   operator Tester*() const
   {
      if (!pointee_) return 0;
      static Tester test;
      return &test;
   }
   ...
};
```

Now if you write `if (sp)`, `operator Tester*` enters into action. This operator returns a null value if and only if `pointee_` is null. `Tester` itself disables operator `delete`, so if somebody calls `delete sp`, a compile-time error occurs. Interestingly, `Tester`'s definition itself lies in the private part of `SmartPtr`, so the client code cannot do anything else with it.

`SmartPtr` addresses the issue of tests for equality and inequality as follows:

- Define `operator==` and `operator!=` in two flavors (templated and nontemplated).
- Define `operator!`.
- If you allow automatic conversion to the pointee type, then define an additional conversion to `void*` to ambiguate a call to the `delete` operator intentionally; otherwise, define a private inner class `Tester` that declares a private `operator delete`, and define a conversion to `Tester*` for `SmartPtr` that returns a null pointer if and only if the pointee object is null.

7.9 Ordering Comparisons

The ordering comparison operators are `operator<`, `operator<=`, `operator>`, and `operator>=`. You can implement them all in terms of `operator<`.

Whether to allow ordering of smart pointers is an interesting question in and of itself and relates to the dual nature of pointers that consistently confuses programmers. Pointers are two concepts in one: iterators and monikers. The iterative nature of pointers allows you to walk through an array of objects using a pointer. Pointer arithmetic, including comparisons, supports this iterative nature of pointers. At the same time, pointers are monikers—inexpensive object representatives that can travel quickly and access the objects in a snap. The dereferencing operators * and -> support the moniker concept.

The two natures of pointers can be confusing at times, especially when you need only one of them. For operating with a vector, you might use both iteration and dereferencing, whereas for walking through a linked list or for manipulating individual objects, you use only dereferencing.

Ordering comparisons for pointers is defined only when the pointers belong to the same contiguous memory. In other words, you can use ordering comparisons only for pointers that point to elements in the same array.

Defining ordering comparisons for smart pointers boils down to this question: Do smart pointers to the objects in the same array make sense? On the face of it, the answer is no. Smart pointers' main feature is to manage object ownership, and objects with separate ownership do not usually belong to the same array. Therefore, it would be dangerous to allow users to make nonsensical comparisons.

If you really need ordering comparisons, you can always use explicit access to the raw pointer. The issue here is, again, to find the safest and most expressive behavior most of the time—not any time.

The previous section concludes that an implicit conversion to a raw pointer type is optional. If `SmartPtr`'s client chooses to allow implicit conversion, the following code compiles:

```
SmartPtr<Something> sp1, sp2;
if (sp1 < sp2) // Converts sp1 and sp2 to raw pointer type,
               // then performs the comparison
...
```

This means that if we want to disable ordering comparisons, we must be proactive, disabling them explicitly. A way of doing this is to declare them and never define them, which means that any use will trigger a link-time error.

```
template <class T>
class SmartPtr
{ ... };

template <class T, class U>
bool operator<(const SmartPtr<T>&, const U&); // Not defined
template <class T, class U>
bool operator<(const T&, const SmartPtr<U>&); // Not defined
```

However, it is wiser to define all other operators in terms of `operator<`, as opposed to leaving them undefined. This way, if `SmartPtr`'s users think it's best to introduce smart pointer ordering, they only have to define `operator<`.

```
// Ambiguity buster
template <class T, class U>
bool operator<(const SmartPtr<T>& lhs, const SmartPtr<U>& rhs)
{
   return lhs < GetImpl(rhs);
}
// All other operators
template <class T, class U>
bool operator>(SmartPtr<T>& lhs, const U& rhs)
{
   return rhs < lhs;
}
... similarly for the other operators ...
```

Note the presence, again, of an ambiguity buster. Now if some library user thinks that SmartPtr<Something> should be ordered, the following code is the ticket:

```
template <class T>
inline bool operator<(const SmartPtr<Something>& lhs,
   const T& rhs)
{
   return GetImpl(lhs) < lhs;
}

template <class T>
inline bool operator<(const T& lhs,
   const SmartPtr<Something>& rhs)
{
   return lhs < GetImpl(rhs);
}
```

It's a pity that the user must define two operators instead of one, but it's so much better than defining eight.

This would conclude the issue of ordering, were it not for an interesting detail. Sometimes it is very useful to have an ordering of arbitrarily located objects, not just objects belonging to the same array. For example, you might need to store supplementary per-object information, and you need to access that information quickly. A map ordered by the address of objects is very effective for such a task.

Standard C++ helps in implementing such designs. Although pointer comparison for arbitrarily located objects is undefined, the standard guarantees that `std::less` yields meaningful results for any two pointers of the same type. Because the standard associative containers use `std::less` as the default ordering relationship, you can safely use maps that have pointers as keys.

`SmartPtr` should support this idiom, too; therefore, `SmartPtr` specializes `std::less`. The specialization simply forwards the call to `std::less` for regular pointers:

```
namespace std
{
   template <class T>
   struct less<SmartPtr<T> >
      : public binary_function<SmartPtr<T>, SmartPtr<T>, bool>
   {
      bool operator()(const SmartPtr<T>& lhs,
         const SmartPtr<T>& rhs) const
      {
         return less<T*>()(GetImpl(lhs), GetImpl(rhs));
      }
      };
   }
```

In summary, SmartPtr does not define ordering operators by default. It declares—without implementing—two generic operator<s and implements all other ordering operators in terms of operator<. The user can define either specialized or generic versions of operator<.

SmartPtr specializes std::less to provide an ordering of arbitrary smart pointer objects.

7.10 Checking and Error Reporting

Applications need various degrees of safety from smart pointers. Some programs are computational-intensive and must be optimized for speed, whereas some others (actually, most) are input/output intensive, which allows better runtime checking without degrading performance.

Most often, right inside an application, you might need both models: low safety/high speed in some critical areas, and high safety/lower speed elsewhere.

We can divide checking issues with smart pointers into two categories: initialization checking and checking before dereference.

7.10.1 Initialization Checking

Should a smart pointer accept the null (zero) value?

A guarantee that a smart pointer cannot be null is easy to implement and may be very useful in practice. It means that any smart pointer is always valid (unless you fiddle with the raw pointer by using GetImplRef). The implementation is easy with the help of a constructor that throws an exception if passed a null pointer.

```
template <class T>
class SmartPtr
{
public:
   SmartPtr(T* p) : pointee_(p)
   {
      if (!p) throw NullPointerException();
   }
   ...
};
```

On the other hand, the null value is a convenient "not a valid pointer" placeholder and can often be useful.

Whether to allow null values affects the default constructor, too. If the smart pointer doesn't allow null values, then how would the default constructor initialize the raw pointer? The default constructor could be lacking, but that would make smart pointers harder to deal with. For example, what should you do when you have a SmartPtr member variable but don't have an appropriate initializer for it at construction time? In conclusion, customizing initialization involves providing an appropriate default value.

7.10.2 Checking Before Dereference

Checking before dereference is important because dereferencing the null pointer engenders undefined behavior. For many applications, undefined behavior is not acceptable, so checking the pointer for validity before dereference is the way to go. Checks before dereference belong to `SmartPtr`'s `operator->` and unary `operator*`.

In contrast to the initialization check, the check before dereference can become a major performance bottleneck in your application, because typical applications use (dereference) smart pointers much more often than they create smart pointer objects. Therefore, you should keep a balance between safety and speed. A good rule of thumb is to start with rigorously checked pointers and remove checks from selected smart pointers as profiling demonstrates a need for it.

Can initialization checking and checking before dereference be conceptually separated? No, because there are links between them. If you enforce strict checking upon initialization, then checking before dereference becomes redundant because the pointer is always valid.

7.10.3 Error Reporting

The only sensible choice for reporting an error is to throw an exception.

You can do something in the sense of avoiding errors. For example, if a pointer is null upon dereference, you can initialize it on the fly. This is a valid and valuable strategy called *lazy initialization*—you construct the value only when you first need it.

If you want to check things only during debugging, you can use the standard `assert` or more sophisticated similar macros. The compiler ignores the tests in release mode, so, assuming you remove all null pointer errors during debugging, you reap both the advantage of checking and that of speed.

`SmartPtr` migrates checking to a dedicated Checking policy. This policy implements checking functions (which can optionally provide lazy initialization) and the error reporting strategy.

7.11 Smart Pointers to `const` and `const` Smart Pointers

Raw pointers allow two kinds of `const`ness: the `const`ness of the pointed-to object and that of the pointer itself. The following is an illustration of these two attributes:

```
const Something* pc = new Something; // points to const object
pc->ConstMemberFunction(); // ok
pc->NonConstMemberFunction(); // error
delete pc; // ok (surprisingly)[4]
Something* const cp = new Something; // const pointer
cp->NonConstMemberFunction(); // ok
cp = new Something; // error, can't assign to const pointer
const Something* const cpc = new Something; // const, points to const
cpc->ConstMemberFunction(); // ok
```

[4]Every once in a while, the question "Why can you apply the `delete` operator to pointers to `const`?" starts a fierce debate in the comp.std.c++ newsgroup. The fact is, for better or worse, the language allows it.

```
cpc->NonConstMemberFunction(); // error
cpc = new Something; // error, can't assign to const pointer
```

The corresponding uses of SmartPtr look like this:

```
// Smart pointer to const object
SmartPtr<const Something> spc(new Something);
// const smart pointer
const SmartPtr<Something> scp(new Something);
// const smart pointer to const object
const SmartPtr<const Something> scpc(new Something);
```

The SmartPtr class template can detect the constness of the pointed-to object either through partial specialization or by using the TypeTraits template defined in Chapter 2. The latter method is preferable because it does not incur source-code duplication as partial specialization does.

The implementation of SmartPtr imitates the semantics of pointers to const objects, const pointers, and the combinations thereof.

7.12 Arrays

In most cases, instead of dealing with heap-allocated arrays and using new[] and delete[], you're better off with std::vector. The standard-provided std::vector class template provides everything that dynamically allocated arrays provide, plus much more. The extra overhead incurred is negligible in most cases.

However, "most cases" is not "always." There are many situations in which you don't need and don't want a full-fledged vector; a dynamically allocated array is exactly what you need. It is awkward in these cases to be unable to exploit smart pointer capabilities. There is a certain gap between the sophisticated std::vector and dynamically allocated arrays. Smart pointers could close that gap by providing array semantics if the user needs them.

From the viewpoint of a smart pointer to an array, the only important issue is to call delete[] pointee_ in its destructor instead of delete pointee_. This issue is already tackled by the Ownership policy.

A secondary issue is providing indexed access, by overloading operator[] for smart pointers. This is technically feasible; in fact, a preliminary version of SmartPtr did provide a separate policy for optional array semantics. However, only in very rare cases do smart pointers point to arrays. In those cases, there already is a way of providing indexed accessing if you use GetImpl:

```
SmartPtr<Widget> sp = ...;
// Access the sixth element pointed to by sp
Widget& obj = GetImpl(sp)[5];
```

It seems like a bad decision to strive for providing extra syntactic convenience at the expense of introducing a new policy.

SmartPtr supports customized destruction via the Ownership policy. You can therefore arrange array-specific destruction via `delete[]`. However, `SmartPtr` does not provide pointer arithmetic.

7.13 Smart Pointers and Multithreading

Most often, smart pointers help with sharing objects. Multithreading issues affect object sharing. Therefore, multithreading issues affect smart pointers.

The interaction between smart pointers and multithreading takes place at two levels. One is the pointee object level, and the other is the bookkeeping data level.

7.13.1 Multithreading at the Pointee Object Level

If multiple threads access the same object and if you access that object through a smart pointer, it can be desirable to lock the object during a function call made through `operator->`. This is possible by having the smart pointer return a proxy object instead of a raw pointer. The proxy object's constructor locks the pointee object, and its destructor unlocks it. The technique is illustrated in Stroustrup (2000). Some code that illustrates this approach is provided here.

First, let's consider a class `Widget` that has two locking primitives, `Lock` and `Unlock`. After a call to `Lock`, you can access the object safely. Any other threads calling `Lock` will block. When you call `Unlock`, you let other threads lock the object.

```
class Widget
{
   ...
   void Lock();
   void Unlock();
};
```

Next, we define a class template `LockingProxy`. Its role is to lock an object (using the `Lock`/`Unlock` convention) for the duration of `LockingProxy`'s lifetime.

```
template <class T>
class LockingProxy
{
public:
   LockingProxy(T* pObj) : pointee_ (pObj)
   { pointee_->Lock(); }
   ~LockingProxy()
   { pointee_->Unlock(); }
   T* operator->() const
   { return pointee_; }
private:
   LockingProxy& operator=(const LockingProxy&);
   T* pointee_;
};
```

In addition to the constructor and destructor, `LockingProxy` defines an `operator->` that returns a pointer to the pointee object.

Although LockingProxy looks somewhat like a smart pointer, there is one more layer to it—the SmartPtr class template itself.

```
template <class T>
class SmartPtr
{
   ...
   LockingProxy<T> operator->() const
   { return LockingProxy<T>(pointee_); }
private:
   T* pointee_;
};
```

Recall from Section 7.3, which explains the mechanics of operator->, that the compiler can apply operator-> multiple times to one -> expression, until it reaches a native pointer. Now imagine you issue the following call (assuming Widget defines a function DoSomething):

```
SmartPtr<Widget> sp = ...;
sp->DoSomething();
```

Here's the trick: SmartPtr's operator-> returns a temporary LockingProxy<T> object. The compiler keeps applying operator->. LockingProxy<T>'s operator-> returns a Widget*. The compiler uses this pointer to Widget to issue the call to DoSomething. During the call, the temporary object LockingProxy<T> is alive and locks the object, which means that the object is safely locked. As soon as the call to DoSomething returns, the temporary Locking-Proxy<T> object is destroyed, so the Widget object is unlocked.

Automatic locking is a good application of smart pointer layering. You can layer smart pointers this way by changing the Storage policy.

7.13.2 Multithreading at the Bookkeeping Data Level

Sometimes smart pointers manipulate data in addition to the pointee object. As you just read in Section 7.5, reference-counted smart pointers share some data—namely the reference count—under the covers. If you copy a reference-counted smart pointer from one thread to another, you end up having two smart pointers pointing to the same reference counter. Of course, they also point to the same pointee object, but that's accessible to the user, who can lock it. In contrast, the reference count is not accessible to the user, so managing it is entirely the responsibility of the smart pointer.

Not only reference-counted pointers are exposed to multithreading-related dangers. Reference-tracked smart pointers (Section 7.5.4) internally hold pointers to each other, which are shared data as well. Reference linking leads to communities of smart pointers, not all of which necessarily belong to the same thread. Therefore, every time you copy, assign, and destroy a reference-tracked smart pointer, you must issue appropriate locking; otherwise, the doubly linked list might get corrupted.

In conclusion, multithreading issues ultimately affect smart pointers' implementation. Let's see how to address the multithreading issue in reference counting and reference linking.

7.13.2.1 Multithreaded Reference Counting

If you copy a smart pointer between threads, you end up incrementing the reference count from different threads at unpredictable times.

As the appendix explains, incrementing a value is not an atomic operation. For incrementing and decrementing integral values in a multithreaded environment, you must use the type `ThreadingModel<T>::IntType` and the `AtomicIncrement` and `AtomicDecrement` functions.

Here things become a bit tricky. Better said, they become tricky if you want to separate reference counting from threading.

Policy-based class design prescribes that you decompose a class into elementary behavioral elements and confine each of them to a separate template parameter. In an ideal world, `SmartPtr` would specify an Ownership policy and a ThreadingModel policy and would use them both toward a correct implementation.

In the case of multithreaded reference counting, however, things are way too tied together. For example, the counter must be of type `ThreadingModel<T>::IntType`. Then, instead of using `operator++` and `operator−−`, you must use `AtomicIncrement` and `AtomicDecrement`. Threading and reference counting melt together; it is unjustifiably hard to separate them.

The best thing to do is to incorporate multithreading in the Ownership policy. Then you can have two implementations: `RefCounting` and `MultiThreadedRefCounting`.

7.13.2.2 Multithreaded Reference Linking

Consider the destructor of a reference-linked smart pointer. It likely looks like this:

```
template <class T>
class SmartPtr
{
public:
   ~SmartPtr()
   {
      if (prev_ == next_)
      {
         delete pointee_;
      }
      else
      {
         prev_->next_ = next_;
         next_->prev_ = prev_;
      }
   }
   ...
private:
   T* pointee_;
   SmartPtr* prev_;
```

```
      SmartPtr* next_;
   };
```

The code in the destructor performs a classic doubly linked list deletion. To make implementation simpler and faster, the list is circular—the last node points to the first node. This way we don't have to test `prev_` and `next_` against zero for any smart pointer. A circular list with only one element has `prev_` and `next_` equal to `this`.

If multiple threads destroy smart pointers that are linked to each other, clearly the destructor must be atomic (uninterruptible by other threads). Otherwise, another thread can interrupt the destructor of a `SmartPtr`, for instance, between updating `prev_->next_` and updating `next_->prev_`. That thread will then operate on a corrupt list.

Similar reasoning applies to `SmartPtr`'s copy constructor and the assignment operator. These functions must be atomic because they manipulate the ownership list.

Interestingly enough, we cannot apply object-level locking semantics here. The appendix divides locking strategies into *class-level* and *object-level* strategies. A class-level locking operation locks all objects in a given class during that operation. An object-level locking operation locks only the object that's subject to that operation. The former technique leads to less memory being occupied (only one mutex per class) but is exposed to performance bottlenecks. The latter is heavier in size (one mutex per object) but might be speedier.

We cannot apply object-level locking to smart pointers because an operation manipulates up to three objects: the current object that's being added or removed, the previous object, and the next object in the ownership list.

If we want to introduce object-level locking, the starting observation is that there must be one mutex per pointee object—because there's one list per pointee object. We can dynamically allocate a mutex for each object, but this nullifies the main advantage of reference linking over reference counting. Reference linking was more appealing exactly because it didn't use the free store.

Alternatively, we can use an intrusive approach: The pointee object holds the mutex, and the smart pointer manipulates that mutex. But the existence of a sound, effective alternative—reference-counted smart pointers—removes the incentive to provide this feature.

In summary, smart pointers that use reference counting or reference linking are affected by multithreading issues. Thread-safe reference counting needs integer atomic operations. Thread-safe reference linking needs mutexes. `SmartPtr` provides only thread-safe reference counting.

7.14 Putting It All Together

Not much to go! Here comes the fun part. So far we have treated each issue in isolation. It's now time to collect all the decisions into a unique `SmartPtr` implementation.

The strategy we'll use is the one described in Chapter 1: policy-based class design. Each design aspect that doesn't have a unique solution migrates to a policy. The `SmartPtr` class template accepts each policy as a separate template parameter. `SmartPtr` inherits all these template parameters, allowing the corresponding policies to store state.

Let's recap the previous sections by enumerating the variation points of `SmartPtr`. Each variation point translates into a policy.

- Storage *policy (Section 7.3).* By default, the stored type is `T*` (`T` is the first template parameter of `SmartPtr`), the pointer type is again `T*`, and the reference type is `T&`. The means of destroying the pointee object is the `delete` operator.
- Ownership *policy (Section 7.5).* Popular implementations are deep copy, reference counting, reference linking, and destructive copy. Note that Ownership is not concerned with the mechanics of destruction itself; this is Storage's task. Ownership controls the *moment* of destruction.
- Conversion *policy (Section 7.7).* Some applications need automatic conversion to the underlying raw pointer type; others do not.
- Checking *policy (Section 7.10).* This policy controls whether an initializer for `SmartPtr` is valid and whether a `SmartPtr` is valid for dereferencing.

Other issues are not worth dedicating separate policies to them or have an optimal solution:

- The address-of operator (Section 7.6) is best not overloaded.
- Equality and inequality tests are handled with the tricks shown in Section 7.8.
- Ordering comparisons (Section 7.9) are left unimplemented; however, Loki specializes `std::less` for `SmartPtr` objects. The user may define an `operator<`, and Loki helps by defining all other ordering comparisons in terms of `operator<`.
- Loki defines `const`-correct implementations for the `SmartPtr` object, the pointee object, or both.
- There is no special support for arrays, but one of the Storage canned implementations can dispose of arrays by using `operator delete[]`.

The presentation of the design issues surrounding smart pointers made these issues easier to understand and more manageable because each issue was discussed in isolation. It would be very helpful, then, if the implementation could decompose and treat issues in isolation instead of fighting with all the complexity at once.

Divide et Impera—this old principle coined by Julius Caesar can be of help even today with smart pointers. (I'd bet money he hadn't predicted that.) We break the problem into small component classes, called *policies.* Each policy class deals with exactly one issue. `SmartPtr` inherits all these classes, thus inheriting all their features. It's that simple—yet incredibly flexible, as you will soon see. Each policy is also a template parameter, which means you can mix and match existing stock policy classes or build your own.

The pointed-to type comes first, followed by each of the policies. Here is the resulting declaration of `SmartPtr`:

```
template
<
   typename T,
   template <class> class OwnershipPolicy = RefCounted,
   class ConversionPolicy = DisallowConversion,
      template <class> class CheckingPolicy = AssertCheck,
      template <class> class StoragePolicy = DefaultSPStorage
   >
   class SmartPtr;
```

The order in which the policies appear in `SmartPtr`'s declaration puts the ones that you customize most often at the top.

The four following subsections discuss the requirements of the four policies we defined. A rule for all policies is that they must have value semantics; that is, they must define a proper copy constructor and assignment operator.

7.14.1 The Storage *Policy*

The Storage policy abstracts the structure of the smart pointer. It provides type definitions and stores the actual `pointee_` object.

If `StorageImpl` is an implementation of the Storage policy and `storageImpl` is an object of type `StorageImpl<T>`, then the constructs in Table 7.1 apply.

Here is the default Storage policy implementation:

```
template <class T>
struct DefaultSPStorage
{
protected:
   typedef T* StoredType;
   typedef T* PointerType;
   typedef T& ReferenceType;
public:
   DefaultSPStorage(StoredType p) : pointee_(p) {}
   PointerType operator->() { return pointee_; }
   ReferenceType operator*() { return pointee_; }
protected:
   void Release()
   {
      delete pointee_;
   }
private:
   T* pointee_;
};
```

In addition to `DefaultSPStorage`, Loki also defines the following:

- `ArrayStorage`, which uses `operator delete[]` inside `Release`
- `LockedStorage`, which uses layering to provide a smart pointer that locks data while dereferenced (see Section 7.13.1)
- `HeapStorage`, which uses an explicit destructor call followed by `std::free` to release the data

Table 7.1: Storage Policy Constructs

Expression	*Semantics*
`StorageImpl<T>::StoredType`	The type actually stored by the implementation. Default: `T*`.
`StorageImpl<T>::PointerType`	The pointer type defined by implementation. This is the type returned by `SmartPtr`'s `operator->`. Default: `T*`. Can be different from `StorageImpl<T>::StoredType` when using smart pointer layering (see Sections 7.3, 7.13.1).
`StorageImpl<T>::ReferenceType`	The reference type. This is the type returned by `SmartPtr`'s `operator*`. Default: `T&`.
`GetImpl(storageImpl)`	Returns an object of type `StorageImpl<T>-::StoredType`.
`GetImplRef(storageImpl)`	Returns an object of type `StorageImpl<T>-::StoredType&`, qualified with `const` if `storageImpl` is `const`.
`storageImpl.operator->()`	Returns an object of type `StorageImpl<T>-::PointerType`. Used by `SmartPtr`'s own `operator->`.
`storageImpl.operator*()`	Returns an object of type `StorageImpl<T>-::ReferenceType`. Used by `SmartPtr`'s own `operator*`.
`StorageImpl<T>::StoredType p;`	Returns the default value (usually zero).
`p = storageImpl.Default();`	
`storageImpl.Destroy()`	Destroys the pointee object.

7.14.2 The Ownership *Policy*

The Ownership policy must support intrusive as well as nonintrusive reference counting. Therefore, it uses explicit function calls rather than constructor/destructor techniques, as Koenig (1996) does. The reason is that you can call member functions at any time, whereas constructors and destructors are called automatically and only at specific times.

The Ownership policy implementation takes one template parameter, which is the corresponding pointer type. `SmartPtr` passes `StoragePolicy<T>::PointerType` to `OwnershipPolicy`. Note that `OwnershipPolicy`'s template parameter is a pointer type, not an object type.

If `OwnershipImpl` is an implementation of Ownership and `ownershipImpl` is an object of type `OwnershipImpl<P>`, then the constructs in Table 7.2 apply.

Table 7.2: Ownership Policy Constructs

Expression	***Semantics***

`P val1;`
`P val2; = OwnershipImplImpl.`
`Clone(val1);`
	Clones an object. It can modify the source value if `OwnershipImpl` uses destructive copy.

`const P val1;`
`P val2 = ownershipImpl.`
`Clone(val1);`
	Clones an object.

`P val;`
`ownershipImpl Release(val);`
	Releases ownership of an object.

`P val;`
`bool unique = ownershipImpl.`
`IsUnique(val);`
	Tests whether a value is uniquely referred. If `IsUnique` returns `true`, then `SmartPtr`'s destructor fires the `Destroy` member function of the Storage policy.

`bool dc = OwnershipImpl<P>`
`::destructiveCopy;`
	States whether `OwnershipImpl` uses destructive copy. If that's the case, `SmartPtr` uses the Colvin/Gibbons trick (Meyers 1999) used in `std::auto_ptr`.

An implementation of Ownership that supports reference counting is shown in the following:

```
template <class P>
class RefCounted
{
   unsigned int*pCount_;
protected:
   RefCounted() : pCount_(new unsigned int(1)) {}
   bool IsUnique const
   {
      return *pCount_ == 1;
   }
   P Clone(const P & val)
   {
      ++*pCount_;
      return val;
   }
   void Release(const P&)
   {
      if (!–*pCount_) delete pCount_;
   }
   enum { destructiveCopy = false }; // see below
};
```

Implementing a policy for reference counting is very easy. Let's write an Ownership policy implementation for COM objects. COM objects have two functions, `AddRef` and `Release`. Upon the last `Release` call, the object destroys itself. You only have to direct `Clone` to `AddRef` and `Release` to COM's `Release`:

```
template <class P>
class COMRefCounted
{
protected:
   static bool IsUnique()const
   {
      return false;
   }
   static P Clone(const P & val)
   {
      val->AddRef();
      return val;
   }
   static void Release(const P&)
   {
      val->Release();
   }
   enum { destructiveCopy = false }; // see below
};
```

Loki defines the following Ownership implementations:

- `DeepCopy`, described in Section 7.5.1. `DeepCopy` assumes the pointee class implements a member function `Clone`.
- `RefCounted`, described in Section 7.5.3 and in this section.
- `RefCountedMT`, a multithreaded version of `RefCounted`.
- `COMRefCounted`, a variant of intrusive reference counting described in this section.
- `RefLinked`, described in Section 7.5.4.
- `DestructiveCopy`, described in Section 7.5.5.
- `NoCopy`, which does not define `Clone`, thus disabling any form of copying.

7.14.3 The Conversion *Policy*

Conversion is a simple policy: It defines a Boolean compile-time constant that says whether `SmartPtr` allows implicit conversion to the underlying pointer type or not.

If `ConversionImpl` is an implementation of Conversion, then the construct in Table 7.3 applies.

The underlying pointer type of `SmartPtr` is dictated by its Storage policy and is `StorageImpl<T>::PointerType`.

As you would expect, Loki defines precisely two Conversion implementations:

- `AllowConversion`
- `DisallowConversion`

Table 7.3: Conversion Policy Construct

Expression	***Semantics***
`bool allowConv = ConversionImpl<P>::allow;`	If `allow` is `true`, `SmartPtr` allows implicit conversion to its underlying pointer type.

Table 7.4: Checking Policy Constructs

Expression	***Semantics***
`S value; checkingImpl.OnDefault(value);`	`SmartPtr` calls `OnDefault` in the default constructor call. If `CheckingImpl` does not define this function, it disables the default constructor at compile time.
`S value; checkingImpl.OnInit(value);`	`SmartPtr` calls `OnInit` upon a constructor call.
`S value; checkingImpl.OnDereference (value);`	`SmartPtr` calls `OnDereference` before returning from `operator->` and `operator*`.
`const S value; checkingImpl.OnDereference (value);`	`SmartPtr` calls `OnDereference` before returning from the `const` versions of `operator->` and `operator*`.

7.14.4 The Checking *Policy*

As discussed in Section 7.10, there are two main places to check a `SmartPtr` object for consistency: during initialization and before dereference. The checks themselves might use `assert`, exception, or lazy initialization or not do anything at all.

The Checking policy operates on the `StoredType` of the Storage policy, not on the `PointerType`. (See Section 7.14.1 for the definition of Storage.)

If `S` is the stored type as defined by the Storage policy implementation, `CheckingImpl` is an implementation of Checking, and `checkingImpl` is an object of type `CheckingImpl<S>`, then the constructs in Table 7.4 apply.

Loki defines the following implementations of Checking:

- `AssertCheck`, which uses `assert` for checking the value before dereferencing.
- `AssertCheckStrict`, which uses `assert` for checking the value upon initialization.
- `RejectNullStatic`, which does not define `OnDefault`. Consequently, any use of `Smart-Ptr`'s default constructor yields a compile-time error.

- `RejectNull`, which throws an exception if you try to dereference a null pointer.
- `RejectNullStrict`, which does not accept null pointers as initializers (again, by throwing an exception).
- `NoCheck`, which handles errors in the grand C and C++ tradition—that is, it does no checking at all.

7.15 Summary

Congratulations! You have just read one of the longest, wildest chapters of this book—an effort that it is hoped paid off. Now you know a lot of things about smart pointers and are equipped with a pretty comprehensive and configurable `SmartPtr` class template.

Smart pointers imitate built-in pointers in syntax and semantics. In addition, they perform a host of tasks that built-in pointers cannot. These tasks might include ownership management and checking against invalid values.

Smart pointer concepts go beyond actual pointer behavior; they can be generalized into smart resources, such as monikers (handles that don't have pointer syntax, yet resemble pointer behavior in the way they enable resource access).

Because they nicely automate things that are very hard to manage by hand, smart pointers are an essential ingredient of successful, robust applications. As small as they are, they can make the difference between a successful project and a failure—or, more often, between a correct program and one that leaks resources like a sieve.

That's why a smart pointer implementer should invest as much attention and effort in this task as possible; the investment is likely to pay in the long term. Similarly, smart pointer users should understand the conventions that smart pointers establish and use them in accordance with those conventions.

The presented implementation of smart pointers focuses on decomposing the areas of functionality into independent policies that the main class template `SmartPtr` mixes and matches. This is possible because each policy implements a well-defined interface.

7.16 `SmartPtr` Quick Facts

- `SmartPtr` declaration:

```
template
<
   typename T,
   template <class> class OwnershipPolicy = RefCounted,
   class ConversionPolicy = DisallowConversion,
   template <class> class CheckingPolicy = AssertCheck,
   template <class> class StoragePolicy = DefaultSPStorage
>
class SmartPtr;
```

- `T` is the type to which `SmartPtr` points. `T` can be a primitive type or a user-defined type. The `void` type is allowed.
- For the remaining class template parameters (`OwnershipPolicy`, `ConversionPolicy`, `CheckingPolicy`, and `StoragePolicy`), you can implement your own policies or choose from the defaults mentioned in Sections 7.14.1 through 7.14.4.

- `OwnershipPolicy` controls the ownership management strategy. You can select from the predefined classes `DeepCopy`, `RefCounted`, `RefCountedMT`, `COMRefCounted`, `RefLinked`, `DestructiveCopy`, and `NoCopy`, described in Section 7.14.2.
- `ConversionPolicy` controls whether implicit conversion to the pointee type is allowed or not. The default is to forbid implicit conversion. Either way, you can still access the pointee object by calling `GetImpl`. You can use the `AllowConversion` and `Disallow-Conversion` implementations (Section 7.14.3).
- `CheckingPolicy` defines the error checking strategy. The defaults provided are `Assert-Check`, `AssertCheckStrict`, `RejectNullStatic`, `RejectNull`, `RejectNullStrict`, and `NoCheck` (Section 7.14.4).
- `StoragePolicy` defines the details of how the pointee object is stored and accessed. The default is `DefaultSPStorage`, which, when instantiated with a type `T`, defines the reference type as `T&`, the stored type as `T*`, and the type returned from `operator->` as `T*` again. Other storage types defined by Loki are `ArrayStorage`, `LockedStorage`, and `HeapStorage` (Section 7.14.1).