significant. On the other hand, access to the contained pointer involves an extra indirection in **f2()** compared to **f()**. This, too, is unlikely to be significant in most programs, so the choice between the styles of **f()** and **f2()** has to be made on reasoning about code quality.

Here is a simple example of a deleter used to provide guaranteed release of data obtained from a C program fragment using **malloc()** (§43.5):

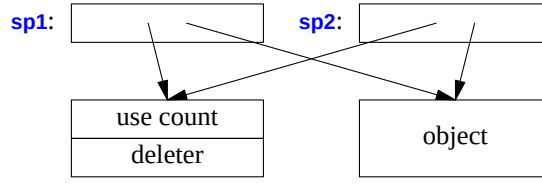
```
extern "C" char* get_data(const char* data);  // get data from C program fragment

using PtoCF = void(*)(void*);

void test()
{
    unique_ptr<char,PtoCF> p {get_data("my_data"),free};
    // ... use *p ...
}   // implicit free(p)
```

Currently, there is no standard-library **make_unique()** similar to **make_pair()** (§34.2.4.1) and **make_shared()** (§34.3.2). However, it is easily defined:

```
template<typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args)        // default deleter version
{
        return unique_ptr<T>{new T{args...}};
}
```

34.3.2 shared_ptr

A **shared_ptr** represents shared ownership. It is used where two pieces of code need access to some data but neither has exclusive ownership (in the sense of being responsible for destroying the object). A **shared_ptr** is a kind of counted pointer where the object pointed to is deleted when the use count goes to zero. Think of a shared pointer as a structure with two pointers: one to the object and one to the use count:



The *deleter* is what is used to delete the shared object when the use count goes to zero. The default deleter is the usual **delete** (invoke the destructor, if any, and deallocate free store).

For example, consider a **Node** in a general graph used by an algorithm that adds and removes both nodes and connections between nodes (edges). Obviously, to avoid resource leaks, a **Node** must be deleted if and only if no other node refers to it. We could try:

```
struct Node {
    vector<Node*> edges;
    // ...
};
```

Given that, answering questions such as ''How many nodes points to this node?'' is very hard and requires much added ''housekeeping'' code. We could plug in a garbage collector (§34.5), but that could have negative performance implications if the graph was only a small part of a large application data space. Worse, if the container contained non-memory resources, such as thread handles, file handles, locks, etc., even a garbage collector would leak resources.

Instead, we can use a **shared_ptr**:

```
struct Node {
    vector<shared_ptr<Node>> edges;
    thread worker;
    // ...
};
```

Here, **Node**'s destructor (the implicitly generated destructor will do fine) **delete**s its **edges**. That is, the destructor for each **edges[i]** is invoked, and the **Node** pointed to (if any) is deleted if **edges[i]** was the last pointer to it.

Don't use a **shared_ptr** just to pass a pointer from one owner to another; that's what **unique_ptr** is for, and **unique_ptr** does it better and more cheaply. If you have been using counted pointers as return values from factory functions (§21.2.4) and the like, consider upgrading to **unique_ptr** rather than **shared_ptr**.

Do not thoughtlessly replace pointers with **shared_ptr**s in an attempt to prevent memory leaks; **shared_ptr**s are not a panacea nor are they without costs:

- A circular linked structure of **shared_ptr**s can cause a resource leak. You need some logical complication to break the circle, for example, use a **weak_ptr** (§34.3.3).
- Objects with shared ownership tend to stay ''live'' for longer than scoped objects (thus causing higher average resource usage).
- Shared pointers in a multi-threaded environment can be expensive (because of the need to prevent data races on the use count).
- A destructor for a shared object does not execute at a predictable time, so the algorithms/logic for the update of any shared object are easier to get wrong than for an object that's not shared. For example, which locks are set at the time of the destructor's execution? Which files are open? In general, which objects are ''live'' and in appropriate states at the (unpredictable) point of execution?
- If a single (last) node keeps a large data structure alive, the cascade of destructor calls triggered by its deletion can cause a significant ''garbage collection delay.'' That can be detrimental to real-time response.

A **shared_ptr** represents shared ownership and can be very useful, even essential, but shared ownership isn't my ideal, and it always carries a cost (independently of how you represent the sharing). It is better (simpler) if an object has a definite owner and a definite, predictable life span. When there is a choice:

- Prefer unique_ptr to shared_ptr.
- Prefer ordinary scoped objects to objects on the heap owned by a unique_ptr.

The shared_ptr provides a fairly conventional set of operations:

shared_ptr<T> Operations (§iso.20.7.2.2) cp is the contained pointer; uc is the use count	
shared_ptr sp {}	Default constructor: cp=nullptr; uc=0; noexcept
shared_ptr sp {p}	Constructor: cp=p; uc=1
shared_ptr sp {p,del}	Constructor: cp=p; uc=1; use deleter del
shared_ptr sp {p,del,a}	Constructor: cp=p; uc=1; use deleter del and allocator a
shared_ptr sp {sp2}	Move and copy constructors: the move constructor moves and then sets sp2.cp=nullptr; the copy constructor copies and sets ++uc for the now-shared uc
sp.~shared_ptr()	Destructor: --uc; delete the object pointed to by cp if uc became 0, using the deleter (the default deleter is delete)
sp=sp2	Copy assignment: ++uc for the now-shared uc; noexcept
sp=move(sp2)	Move assignment: sp2.cp=nullptr for the now-shared uc; noexcept
bool b {sp};	Conversion to bool: sp.uc==nullptr; explicit
sp.reset()	shared_ptr{}.swap(sp); that is, sp contains pointer{}, and the destruction of the temporary shared_ptr{} decreases the use count for the old object; noexcept
sp.reset(p)	shared_ptr{p}.swap(sp); that is, sp.cp=p; uc==1; the destruction of the temporary shared_ptr decreases the use count for the old object
sp.reset(p,d)	Like sp.reset(p) but with the deleter d
sp.reset(p,d,a)	Like sp.reset(p) but with the deleter d and the allocator a
p=sp.get()	p=sp.cp; noexcept
x=*sp	x=*sp.cp; noexcept
x=sp->m	x=sp.cp->m; noexcept
n=sp.use_count()	n is the value of the use count (0 if sp.cp==nullptr)
sp.unique()	sp.uc==1? (does not check if sp.cp==nullptr)
x=sp.owner_before(pp)	x is an ordering function (strict weak order; §31.2.2.1) pp is a shared_ptr or a weak_ptr
sp.swap(sp2)	Exchange sp's and sp2's values; noexcept

In addition, the standard library provides a few helper functions:

shared_ptr<T> Helpers (§iso.20.7.2.2.6, §iso.20.7.2.2.7) (continues)	
sp=make_shared(args)	sp is a shared_ptr<T> for an object of type T constructed from the arguments args; allocated using new
sp=allocate_shared(a,args)	sp is a shared_ptr<T> for an object of type T constructed from the arguments args; allocated using allocator a

shared_ptr<T> Helpers (continued) (§iso.20.7.2.2.6, §iso.20.7.2.2.7)	
sp==sp2	sp.cp==sp2.cp; sp or sp2 may be the nullptr
sp<sp2	less<T*>(sp.cp,sp2.cp); sp or sp2 may be the nullptr
sp!=sp2	!(sp=sp2)
sp>sp2	sp2<sp
sp<=sp2	!(sp>sp2)
sp>=sp2	!(sp<sp2)
swap(sp,sp2)	sp.swap(sp2)
sp2=static_pointer_cast(sp)	static_cast for shared pointers: sp2=shared_ptr<T>(static_cast<T*>(sp.cp)); noexcept
sp2=dynamic_pointer_cast(sp)	dynamic_cast for shared pointers: sp2=shared_ptr<T>(dynamic_cast<T*>(sp.cp)); noexcept
sp2=const_pointer_cast(sp)	const_cast for shared pointers: sp2=shared_ptr<T>(const_cast<T*>(sp.cp)); noexcept
dp=get_deleter<D>(sp)	If sp has a deleter of type D, *dp is sp's deleter; otherwise, dp==nullptr; noexcept
os<<sp	Write sp to ostream os

For example:

```
struct S {
    int i;
    string s;
    double d;
    // ...
};

auto p = make_shared<S>(1,"Ankh Morpork",4.65);
```

Now, p is a shared_ptr<S> pointing to an object of type S allocated on the free store, containing {1,string{"Ankh Morpork"},4.65}.

Note that unlike unique_ptr::get_deleter(), shared_ptr's deleter is not a member function.

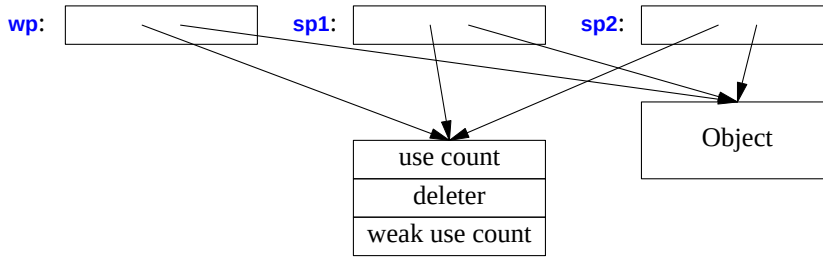
34.3.3 weak_ptr

A weak_ptr refers to an object managed by a shared_ptr. To access the object, a weak_ptr can be converted to a shared_ptr using the member function lock(). A weak_ptr allows access to an object, owned by someone else, that

- You need access to (only) if it exists
- May get deleted (by someone else) at any time
- Must have its destructor called after its last use (usually to delete a non-memory resource)

In particular, we use weak pointers to break loops in data structures managed using shared_ptrs.

Think of a weak_ptr as a structure with two pointers: one to the (potentially shared) object and one to the use count structure of that object's shared_ptrs:



The ''weak use count'' is needed to keep the use count structure alive because there may be `weak_ptr`s after the last `shared_ptr` for an object (and the object) is destroyed.

```
template<typename T>
class weak_ptr {
public:
    using element_type = T;
    // ...
};
```

A `weak_ptr` has to be converted to a `shared_ptr` to access ''its'' object, so it provides relatively few operations:

`weak_ptr<T>` (§iso.20.7.2.3) `cp` is the contained pointer; `wuc` is the weak use count	
`weak_ptr wp {};`	Default constructor: `cp=nullptr`; constexpr; noexcept
`weak_ptr wp {pp};`	Copy constructor: `cp=pp.cp; ++wuc;` `pp` is a `weak_ptr` or a `shared_ptr`; noexcept
`wp.~weak_ptr()`	Destructor: no effect on `*cp`; `--wuc`
`wp=pp`	Copy: decrease `wuc` and set `wp` to `pp`: `weak_ptr(pp).swap(wp);` `pp` is a `weak_ptr` or a `shared_ptr`; noexcept
`wp.swap(wp2)`	Exchange `wp`'s and `wp2`'s values; noexcept
`wp.reset()`	Decrease `wuc` and set `wp` to `nullptr`: `weak_ptr{}.swap(wp)`; noexcept
`n=wp.use_count()`	`n` is the number of `shared_ptr`s to `*cp`; noexcept
`wp.expired()`	Are there any `shared_ptr`s to `*cp`? noexcept
`sp=wp.lock()`	Make a new `shared_ptr` for `*cp`; noexcept
`x=wp.owner_before(pp)`	`x` is an ordering function (strict weak order; §31.2.2.1); `pp` is a `shared_ptr` or a `weak_ptr`
`swap(wp,wp2)`	`wp.swap(wp2)`; noexcept

Consider an implementation of the old ''asteroid game.'' All asteroids are owned by ''the game,'' but each asteroid must keep track of neighboring asteroids and handle collisions. A collision typically leads to the destruction of one or more asteroids. Each asteroid must keep a list of other asteroids in its neighborhood. Note that being on such a neighbor list should not keep an asteroid ''alive'' (so a `shared_ptr` would be inappropriate). On the other hand, an asteroid must not be

destroyed while another asteroid is looking at it (e.g., to calculate the effect of a collision). And obviously, an asteroid destructor must be called to release resources (such as a connection to the graphics system). What we need is a list of asteroids that might still be intact and a way of ''grabbing onto one'' for a while. A **weak_ptr** does just that:

```
void owner()
{
    // ...
    vector<shared_ptr<Asteroid>> va(100);
    for (int i=0; i<va.size(); ++i) {
        // ... calculate neighbors for new asteroid ...
        va[i].reset(new Asteroid(weak_ptr<Asteroid>(va[neighbor]));
        launch(i);
    }
    // ...
}
```

Obviously, I radically simplified ''the owner'' and gave each new **Asteroid** just one neighbor. The key is that we give the **Asteroid** a **weak_ptr** to that neighbor. The owner keeps a **shared_ptr** to represent the ownership that's shared whenever an **Asteroid** is looking (but not otherwise). The collision calculation for an **Asteroid** will look something like this:

```
void collision(weak_ptr<Asteroid> p)
{
    if (auto q = p.lock()) {      // p.lock returns a shared_ptr to p's object
        // ... that Asteroid still existed: calculate ...
    }
    else {      // Oops: that Asteroid has already been destroyed
        p.reset();
    }
}
```

Note that even if a user decides to shut down the game and deletes all **Asteroid**s (by destroying the **shared_ptr**s representing ownership), every **Asteroid** that is in the middle of calculating a collision still finishes correctly: after the **p.lock()**, it holds a **shared_ptr** that will not become invalid.

34.4 Allocators

The STL containers (§31.4) and **string** (Chapter 36) are resource handles that acquire and release memory to hold their elements. To do so, they use *allocators*. The basic purpose of an allocator is to provide a source of memory for a given type and a place to return that memory to once it is no longer needed. Thus, the basic allocator functions are:

```
p=a.allocate(n);      // acquire space for n objects of type T
a.deallocate(p,n);    // release space for n objects of type T pointed to by p
```

For example: