---

### 5.2. Smart Pointers

Since C, we know that pointers are important but are a source of trouble. One reason to use pointers is to have reference semantics outside the usual boundaries of scope. However, it can be very tricky to ensure that their lifetime and the lifetime of the objects they refer to match, especially when multiple pointers refer to the same object. For example, to have the same object in multiple collections (see Chapter 7), you have to pass a pointer into each collection, and ideally there should be no problems when one of the pointers gets destroyed (no "dangling pointers" or multiple deletions of the referenced object) and when the last reference to an object gets destroyed (no "resource leaks").

A usual approach to avoid these kinds of problems is to use "smart pointers." They are "smart" in the sense that they support programmers in avoiding problems such as those just described. For example, a smart pointer can be so smart that it "knows" whether it is the last pointer to an object and uses this knowledge to delete an associated object only when it, as "last owner" of an object, gets destroyed.

Note, however, that it is not sufficient to provide only one smart pointer class. Smart pointers can be smart about different aspects and might fulfill different priorities, because you might pay a price for the smartness. Note that with a specific smart pointer, it's still possible to misuse a pointer or to program erroneous behavior.

Since C++11, the C++ standard library provides two types of smart pointer:

1. Class `shared_ptr` for a pointer that implements the concept of *shared ownership*. Multiple smart pointers can refer to the same object so that the object and its associated resources get released whenever the last reference to it gets destroyed. To perform this task in more complicated scenarios, helper classes, such as `weak_ptr`, `bad_weak_ptr`, and `enable_shared_from_this`, are provided.
2. Class `unique_ptr` for a pointer that implements the concept of *exclusive ownership* or *strict ownership*. This pointer ensures that only one smart pointer can refer to this object at a time. However, you can transfer ownership. This pointer is especially useful for avoiding resource leaks, such as missing calls of `delete` after or while an object gets created with `new` and an exception occurred.

Historically, C++98 had only one smart pointer class provided by the C++ standard library, class `auto_ptr<>`, which was designed to perform the task that `unique_ptr` now provides. However, due to missing language features, such as move semantics for constructors and assignment operators and other flaws, this class turned out to be difficult to understand and error prone. So, after class `shared_ptr` was introduced with TR1 and class `unique_ptr` was introduced with C++11, class `auto_ptr` officially became deprecated with C++11, which means that you should not use it unless you have old existing code to compile.

All smart pointer classes are defined in the `<memory>` header file.

## 5.2.1. Class `shared_ptr`

Almost every nontrivial program needs the ability to use or deal with objects at multiple places at the same time. Thus, you have to "refer" to an object from multiple places in your program. Although the language provides references and pointers, this is not enough, because you often have to ensure that when the last reference to an object gets deleted, the object itself gets deleted, which might require some cleanup operations, such as freeing memory or releasing a resource.

So we need a semantics of "cleanup when the object is nowhere used anymore." Class `shared_ptr` provides this semantics of *shared ownership*. Thus, multiple `shared_ptr`s are able to share, or "own," the same object. The last owner of the object is responsible for destroying it and cleaning up all resources associated with it.

By default, the cleanup is a call of `delete`, assuming that the object was created with `new`. But you can (and often must) define other ways to clean up objects. You can define your own *destruction policy*. For example, if your object is an array allocated with `new[]`, you have to define that the cleanup performs a `delete[]`. Other examples are the deletion of associated resources, such as handles, locks, associated temporary files, and so on.

To summarize, the goal of `shared_ptr`s is to automatically release resources associated with objects when those objects are no longer needed (but not before).

#### Using Class `shared_ptr`

You can use a `shared_ptr` just as you would any other pointer. Thus, you can assign, copy, and compare shared pointers, as well as use operators `*` and `->`, to access the object the pointer refers to. Consider the following example:

```
// util/sharedptr1.cpp

#include <iostream>
#include <string>
#include <vector>
#include <memory>
using namespace std;
```

```
int main()
{
    // two shared pointers representing two persons by their name
    shared_ptr<string> pNico(new string("nico"));
    shared_ptr<string> pJutta(new string("jutta"));

    // capitalize person names
    (*pNico)[0] = 'N';
    pJutta->replace(0,1,"J");

    // put them multiple times in a container
    vector<shared_ptr<string>> whoMadeCoffee;
    whoMadeCoffee.push_back(pJutta);
    whoMadeCoffee.push_back(pJutta);
    whoMadeCoffee.push_back(pNico);
    whoMadeCoffee.push_back(pJutta);
    whoMadeCoffee.push_back(pNico);

    // print all elements
    for (auto ptr : whoMadeCoffee) {
        cout << *ptr << "  ";
    }
    cout << endl;

    // overwrite a name again
    *pNico = "Nicolai";

    // print all elements again
    for (auto ptr : whoMadeCoffee) {
        cout << *ptr << "  ";
    }
    cout << endl;

    // print some internal data
    cout << "use_count: " << whoMadeCoffee[0].use_count() << endl;
}
```

After including `<memory>`, where `shared_ptr` class is defined, two `shared_ptr`s representing pointers to strings are declared and initialized:

```
shared_ptr<string> pNico(new string("nico"));
shared_ptr<string> pJutta(new string("jutta"));
```

Note that because the constructor taking a pointer as single argument is `explicit`, you can't use the assignment notation here because that is considered to be an implicit conversion. However, the new initialization syntax is also possible:

```
shared_ptr<string> pNico = new string("nico");   // ERROR
shared_ptr<string> pNico{new string("nico")};    // OK
```

You can also use the convenience function `make_shared()` here:

```
shared_ptr<string> pNico = make_shared<string>("nico");
shared_ptr<string> pJutta = make_shared<string>("jutta");
```

This way of creation is faster and safer because it uses one instead of two allocations: one for the object and one for the shared data the shared pointer uses to control the object (see Section 5.2.4, page 95, for details).

Alternatively, you can declare the shared pointer first and assign a new pointer later on. However, you can't use the assignment operator; you have to use `reset()` instead:

```
shared_ptr<string> pNico4;
pNico4 = new string("nico");   // ERROR: no assignment for ordinary pointers
pNico4.reset(new string("nico"));   // OK
```

The following two lines demonstrate that using shared pointers is just like using ordinary pointers:

```
(*pNico)[0] = 'N';
pJutta->replace(0,1,"J");
```

With operator `*`, you yield the object `pNico` refers to to assign a capital `'N'` to its first character. With operator `->`, you get access to a member of the object `pJutta` refers to. Thus, here the member function `replace()` allows you to replace substrings (see Section 13.3.7, page 706).

Next we insert both pointers multiple times into a container of type `vector<>` (see Section 7.3, page 270). The container usually creates its own copy of the elements passed, so we would insert copies of strings if we inserted the strings directly. However, because we pass pointers to the strings, these pointers are copied, so the container now contains multiple references to the same object. This means that

if we modify the objects, all occurrences of this object in the container get modified. Thus, after replacing the value of the string `pNico`

```
*pNico = "Nicolai";
```

all occurrences of this object now refer to the new value, as you can see in Figure 5.1 and according to the corresponding output of the program:

```
Jutta  Jutta  Nico  Jutta  Nico
Jutta  Jutta  Nicolai  Jutta  Nicolai
use_count: 4
```

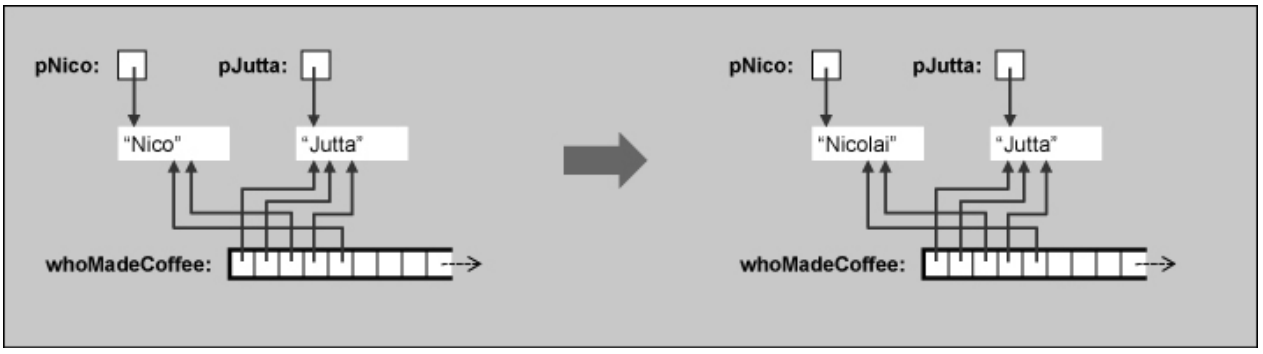


**Figure 5.1. Using *shared_ptr*s**

The last row of the output is the result of calling `use_count()` for the first shared pointer in the vector. `use_count()` yields the current number of owners an object referred to by shared pointers has. As you can see, we have four owners of the object referred to by the first element in the vector: `pJutta` and the three copies of it inserted into the container.

At the end of the program, when the last owner of a string gets destroyed, the shared pointer calls `delete` for the object it refers to. Such a deletion does not necessarily have to happen at the end of the scope. For example, assigning the `nullptr` (see Section 3.1.1, page 14) to `pNico` or resizing the vector so that it contains only the first two elements would delete the last owner of the string initialized with `nico` .

#### Defining a Deleter

We can declare our own deleter, which, for example, prints a message before it deletes the referenced object:

```
shared_ptr<string> pNico(new string("nico"),
                         [](string* p) {
                             cout << "delete " << *p << endl;
                             delete p;
                         });
...
pNico = nullptr;          // pNico does not refer to the string any longer
whoMadeCoffee.resize(2);  // all copies of the string in pNico are destroyed
```

Here, we pass a lambda (see Section 3.1.10, page 28) as the second argument to the constructor of a `shared_ptr` . Having `pNico` declared that way, the lambda function gets called when the last owner of a string gets destroyed. So the preceding program with this modification would print

```
delete Nicolai
```

when `resize()` gets called after all statements as discussed before. The effect would be the same if we first changed the size of the vector and then assigned `nullptr` or another object to `pNico` .

For another example application of `shared_ptr<>` , see how elements can be shared by two containers in Section 7.11, page 388.

#### Dealing with Arrays

Note that the default deleter provided by `shared_ptr` calls `delete` , not `delete[]` . This means that the default deleter is appropriate only if a shared pointer owns a single object created with `new` . Unfortunately, creating a `shared_ptr` for an array is possible but wrong:

```
std::shared_ptr<int> p(new int[10]);  // ERROR, but compiles
```

So, if you use `new[]` to create an array of objects you have to define your own deleter. You can do that by passing a function, function object, or lambda, which calls `delete[ ]` for the passed ordinary pointer. For example:

```
std::shared_ptr<int> p(new int[10],
                       [](int* p) {
```

```
        delete[] p;
    });
```

You can also use a helper officially provided for `unique_ptr`, which calls `delete[]` as deleter (see Section 5.2.5, page 106):

```
std::shared_ptr<int> p(new int[10],
                       std::default_delete<int[]>());
```

Note, however, that `shared_ptr` and `unique_ptr` deal with deleters in slightly different ways. For example, `unique_ptr`s provide the ability to own an array simply by passing the corresponding element type as template argument, whereas for `shared_ptr`s this is not possible:

**Click here to view code image**

```
std::unique_ptr<int[]> p(new int[10]);  // OK
std::shared_ptr<int[]> p(new int[10]);  // ERROR: does not compile
```

In addition, for `unique_ptr`s, you have to specify a second template argument to specify your own deleter:

```
std::unique_ptr<int,void(*)(int*)> p(new int[10],
                                     [](int* p) {
                                         delete[] p;
                                     });
```

Note also that `shared_ptr` does not provide an operator `[]`. For `unique_ptr`, a partial specialization for arrays exists, which provides operator `[]` instead of operators `*` and `->`. The reason for this difference is that `unique_ptr` is optimized for performance and flexibility. See Section 5.2.8, page 114, for details.

#### Dealing with Other Destruction Policies

When the cleanup after the last owning shared pointer is something other than deleting memory, you have to specify your own deleter. You can understand this to specify your own *destruction policy*.

As a first example, suppose that we want to ensure that a temporary file gets removed when the last reference to it gets destroyed. This is how it could be done:

**Click here to view code image**

```
// util/sharedptr2.cpp

#include <string>
#include <fstream>    // for ofstream
#include <memory>     // for shared_ptr
#include <cstdio>     // for remove()

class FileDeleter
{
    private:
      std::string filename;
    public:
      FileDeleter (const std::string& fn)
       : filename(fn) {
      }
      void operator () (std::ofstream* fp) {
          delete fp;                        // close file
          std::remove(filename.c_str()); // delete file
      }
};

int main()
{
    // create and open temporary file:
    std::shared_ptr<std::ofstream> fp(new std::ofstream("tmpfile.txt"),
                                      FileDeleter("tmpfile.txt"));
    ...
}
```

Here, we initialize a `shared_ptr` with a `new`ly created output file (see Section 15.9, page 791). The passed `FileDeleter` ensures that this files gets closed and deleted with the standard C function `remove()`, provided in `<cstdio>` when the last copy of this shared pointer loses the ownership of this output stream. Because `remove()` needs the filename, we pass this as an argument to the constructor of `FileDeleter`.

Our second example demonstrates how to use `shared_ptr`s to deal with shared memory:[8]

[8] There are multiple system-dependent ways to deal with shared memory. Here, the standard POSIX

way with `shm_open()` and `mmap()` is used, which requires `shm_unlink()` to be called to release the (persistent) shared memory.

**Click here to view code image**

```
// util/sharedptr3.cpp

#include <memory>      // for shared_ptr
#include <sys/mman.h> // for shared memory
#include <fcntl.h>
#include <unistd.h>
#include <cstring>     // for strerror()
#include <cerrno>      // for errno
#include <string>
#include <iostream>

class SharedMemoryDetacher
{
  public:
    void operator () (int* p) {
        std::cout << "unlink /tmp1234" << std::endl;
        if (shm_unlink("/tmp1234") != 0) {
            std::cerr << "OOPS: shm_unlink() failed" << std::endl;
        }
    }
};

std::shared_ptr<int> getSharedIntMemory (int num)
{
    void* mem;
    int shmfd = shm_open("/tmp1234", O_CREAT|O_RDWR, S_IRWXU|S_IRWXG);
    if (shmfd < 0) {
        throw std::string(strerror(errno));
    }
    if (ftruncate(shmfd, num*sizeof(int)) == -1) {
        throw std::string(strerror(errno));
    }
    mem = mmap(nullptr, num*sizeof(int), PROT_READ | PROT_WRITE,
               MAP_SHARED, shmfd, 0);
    if (mem == MAP_FAILED) {
        throw std::string(strerror(errno));
    }
    return std::shared_ptr<int>(static_cast<int*>(mem),
                                SharedMemoryDetacher());
}

int main()
{
    // get and attach shared memory for 100 ints:
    std::shared_ptr<int> smp(getSharedIntMemory(100));

    // init the shared memory
    for (int i=0; i<100; ++i) {
        smp.get()[i] = i*42;
    }

    // deal with shared memory somewhere else:
    ...
    std::cout << "<return>" << std::endl;
    std::cin.get();

    // release shared memory here:
    smp.reset();
    ...
}
```

First, a deleter `SharedMemoryDetacher` is defined to detach shared memory. The deleter releases the shared memory, which `getSharedIntMemory()` gets and attaches. To ensure that the deleter is called when the last use of the shared memory is over, it is passed when `getSharedIntMemory()` creates a `shared_ptr` for the attached memory:

**Click here to view code image**

```
return std::shared_ptr<int>(static_cast<int*>(mem),
                            SharedMemoryDetacher());  // calls shmdt()
```

Alternatively, you could use a lambda here (skipping prefix `std::`):

**Click here to view code image**

```
    return shared_ptr<int>(static_cast<int*>(mem),
                           [](int* p) {
                               cout << "unlink /tmp1234" << endl;
                               if (shm_unlink("/tmp1234") != 0) {
                                   cerr << "OOPS: shm_unlink() failed"
                                        << endl;
                               }
                           });
```

Note that the passed deleter is not allowed to throw exceptions. Therefore, we only write an error message to `std::cerr` here.

Because the signature of `shm_unlink()` already fits, you could even use `shm_unlink()` directly as deleter if you don't want to check its return value:

```
    return std::shared_ptr<int>(static_cast<int*>(mem),
                                shm_unlink);
```

Note that `shared_ptr` s provide only operators `*` and `->` . Pointer arithmetic and operator `[]` are not provided. Thus, to access the memory, you have to use `get()` , which yields the internal pointer wrapped by `shared_ptr` to provide the full pointer semantics:

```
    smp.get()[i] = i*42;
```

Thus, `get()` provides an alternative of calling:

```
    (&*smp)[i] = i*42;
```

For both examples, another possible implementation technique is probably cleaner than this: Just create a new class, where the constructor does the initial stuff and the destructor does the cleanup. You can then just use `shared_ptr` s to manage objects of this class created with `new` . The benefit is that you can define a more intuitive interface, such as an operator `[]` for an object representing shared memory. However, you should then carefully think about copy and assignment operations; if in doubt, disable them.

### 5.2.2. Class `weak_ptr`

The major reason to use `shared_ptr` s is to avoid taking care of the resources a pointer refers to. As written, `shared_ptr` s are provided to automatically release resources associated with objects no longer needed.

However, under certain circumstances, this behavior doesn't work or is not what is intended:

- One example is cyclic references. If two objects refer to each other using `shared_ptr` s, and you want to release the objects and their associated resource if no other references to these objects exist, `shared_ptr` won't release the data, because the `use_count()` of each object is still `1` . You might want to use ordinary pointers in this situation, but doing so requires explicitly caring for and managing the release of associated resources.
- Another example occurs when you explicitly want to share but not own an object. Thus, you have the semantics that the lifetime of a reference to an object outlives the object it refers to. Here, `shared_ptr` s would never release the object, and ordinary pointers might not notice that the object they refer to is not valid anymore, which introduces the risk of accessing released data.

For both cases, class `weak_ptr` is provided, which allows sharing but not owning an object. This class requires a shared pointer to get created. Whenever the last shared pointer owning the object loses its ownership, any weak pointer automatically becomes empty. Thus, besides default and copy constructors, class `weak_ptr` provides only a constructor taking a `shared_ptr` .

You can't use operators `*` and `->` to access a referenced object of a `weak_ptr` directly. Instead, you have to create a shared pointer out of it. This makes sense for two reasons:

1. Creating a shared pointer out of a weak pointer checks whether there is (still) an associated object. If not, this operation will throw an exception or create an empty shared pointer (what exactly happens depends on the operation used).
2. While dealing with the referenced object, the shared pointer can't get released.

As a consequence, class `weak_ptr` provides only a small number of operations: Just enough to create, copy, and assign a weak pointer and convert it into a shared pointer or check whether it refers to an object.

#### Using Class `weak_ptr`

Consider the following example:

**Click here to view code image**

```
    // util/weakptr1.cpp

    #include <iostream>
    #include <string>
    #include <vector>
    #include <memory>
    using namespace std;
```

```
class Person {
  public:
    string name;
    shared_ptr<Person> mother;
    shared_ptr<Person> father;
    vector<shared_ptr<Person>> kids;

    Person (const string& n,
            shared_ptr<Person> m = nullptr,
            shared_ptr<Person> f = nullptr)
     : name(n), mother(m), father(f) {
    }

    ~Person() {
      cout << "delete " << name << endl;
    }
};

shared_ptr<Person> initFamily (const string& name)
{
    shared_ptr<Person> mom(new Person(name+"'s mom"));
    shared_ptr<Person> dad(new Person(name+"'s dad"));
    shared_ptr<Person> kid(new Person(name,mom,dad));
    mom->kids.push_back(kid);
    dad->kids.push_back(kid);
    return kid;
}

int main()
{
    shared_ptr<Person> p = initFamily("nico");

    cout << "nico's family exists" << endl;
    cout << "- nico is shared " << p.use_count() << " times" << endl;
    cout << "- name of 1st kid of nico's mom: "
         << p->mother->kids[0]->name << endl;

    p = initFamily("jim");
    cout << "jim's family exists" << endl;
}
```

Here, a class `Person` has a name and optional references to other `Person`s, namely, the parents (`mother` and `father`) and the kids (a vector; see Section 7.3, page 270).

First, `initFamily()` creates three `Person`s: `mom`, `dad`, and `kid`, initialized with corresponding names based on the passed argument. In addition, `kid` is initialized with the parents, and for both parents, `kid` is inserted in the list of kids. Finally, `initFamily()` returns the `kid`. Figure 5.2 shows the resulting situation at the end of `initFamily()` and after calling and assigning the result to `p`.

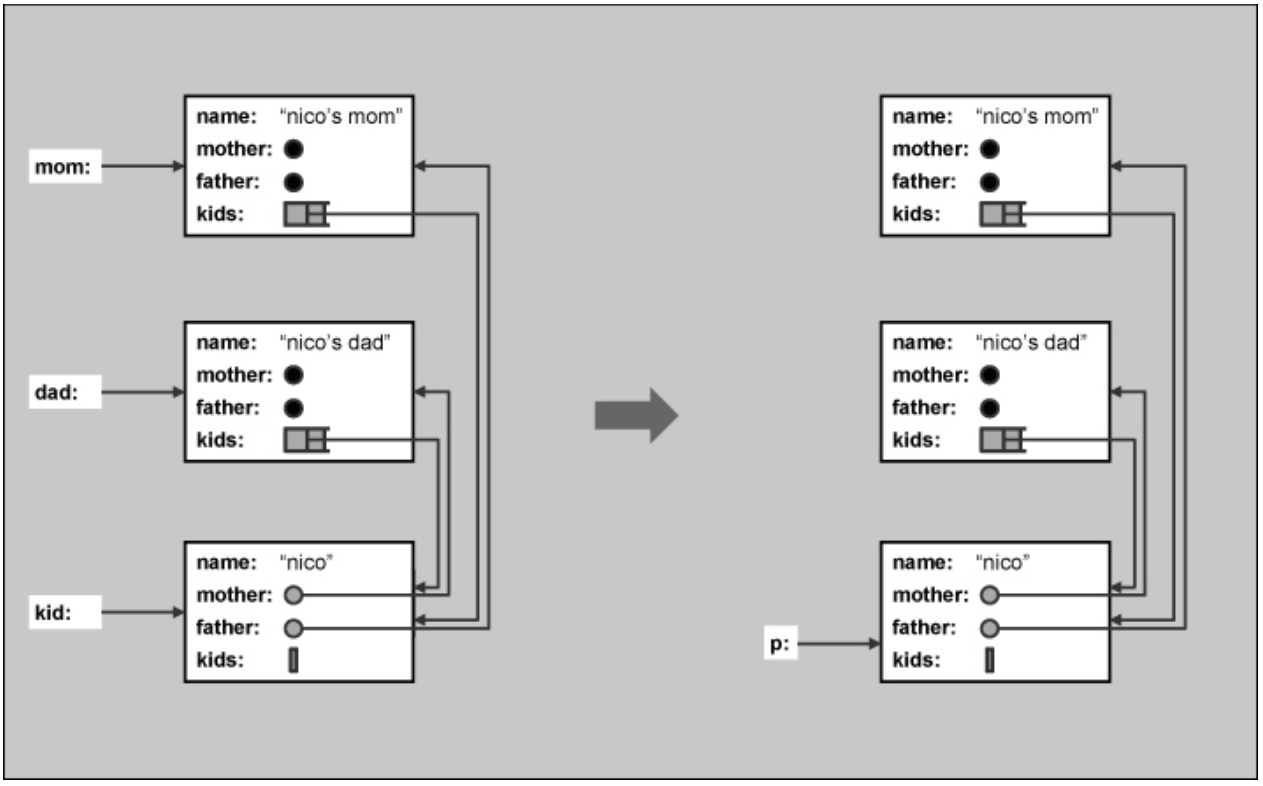


**Figure 5.2. A Family Using `shared_ptr`s Only**

As you can see, `p` is our last handle into the family created. Internally, however, each object has references from the kid to each parent and backwards. So, for example, `nico` was shared three times before `p` gets a new value. Now, if we release our last handle to the family — either by assigning a new person or `nullptr` to `p` or by leaving the scope of `p` at the end of `main()` — none of the `Person`s gets released, because each still has at least one shared pointer referring to it. As a result, the destructor of each `Person`, which would print "`delete` *name*," never gets called:

```
nico's family exists
- nico is shared 3 times
- name of 1st kid of nico's mom: nico
jim's family exists
```

Using `weak_ptr`s instead helps here. For example, we can declare `kids` to be a vector of `weak_ptr`s:

```
// util/weakptr2.cpp
...
class Person {
  public:
    string name;
    shared_ptr<Person> mother;
    shared_ptr<Person> father;
    vector<weak_ptr<Person>> kids;   // weak pointer !!!
    Person (const string& n,
            shared_ptr<Person> m = nullptr,
            shared_ptr<Person> f = nullptr)
        : name(n), mother(m), father(f) {
    }
    ~Person() {
      cout << "delete " << name << endl;
    }
};
...
```

By doing so, we can break the cycle of shared pointers so that in one direction (from kid to parent) a shared pointer is used, whereas from a parent to the kids, weak pointers are used (the dashed line in Figure 5.3).

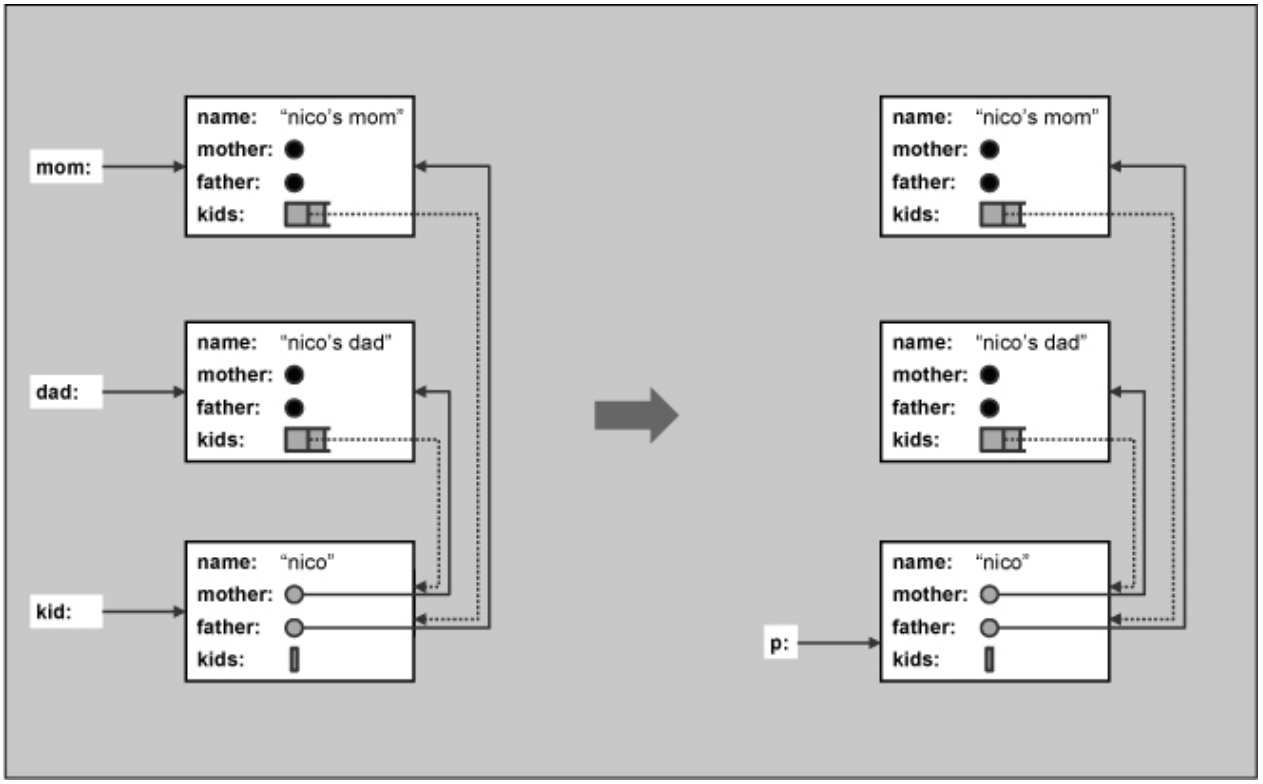


**Figure 5.3. A Family Using *shared_ptr*s and *weak_ptr*s**

As a result, the program now has the following output:

```
nico's family exists
- nico is shared 1 times
- name of 1st kid of nico's mom: nico
delete nico
delete nico's dad
delete nico's mom
jim's family exists
delete jim
delete jim's dad
delete jim's mom
```

As soon as we lose our handle into a family created — either by assigning a new value to `p` or by leaving `main()` — the `kid` 's object of the family loses its last owner ( `use_count()` yielded `1` before), which has the effect that both parents lose their last owner. So all objects, initially created by `new` , are `delete` d now so that their destructors get called.

Note that by using weak pointers, we had to slightly modify the way we access the object referred to via a weak pointer. Instead of calling

```
p->mother->kids[0]->name
```

we now have to insert `lock()` into the expression

```
p->mother->kids[0].lock()->name
```

which yields a `shared_ptr` out of the `weak_ptr` the vector of `kids` contains. If this modification is not possible — for example, because the last owner of the object released the object in the meantime — `lock()` yields an empty `shared_ptr` . In that case, calling operator `*` or `->` would cause undefined behavior.

If you are not sure that the object behind a weak pointer still exists, you have several options:

1. You can call `expired()` , which returns `true` if the `weak_ptr` doesn't share an object any longer. This option is equivalent to checking whether `use_count()` is equal to `0` but might be faster.
2. You can explicitly convert a `weak_ptr` into a `shared_ptr` by using a corresponding `shared_ptr` constructor. If there is no valid referenced object, this constructor will throw a `bad_weak_ptr` exception. This is an exception of a class derived from `std::exception` , where `what()` yields `"bad_weak_ptr"` .[9] See Section 4.3.1, page 43, for details about all standard exceptions.

[9] For exceptions, the return value of `what()` is usually implementation specific. However, the standard specifies to yield `"bad_weak_ptr"` here. Nevertheless, implementations might not follow this advice; for example, GCC 4.6.1 returned `"std::bad_weak_ptr"` .

3. You can call `use_count()` to ask for the number of owners the associated object has. If the return value is `0`, there is no valid object any longer. Note, however, that you should usually call `use_count()` only for debugging purposes; the C++ standard library explicitly states: "`use_count()` is not necessarily efficient."

For example:

Click here to view code image

```
try {
   shared_ptr<string> sp(new string("hi"));     // create shared pointer
   weak_ptr<string> wp = sp;                    // create weak pointer out of it
   sp.reset();                                  // release object of shared
pointer
   cout << wp.use_count() << endl;              // prints: 0
   cout << boolalpha << wp.expired() << endl;   // prints: true
   shared_ptr<string> p(wp);                    // throws std::bad_weak_ptr
}
catch (const std::exception& e) {
   cerr << "exception: " << e.what() << endl;  // prints: bad_weak_ptr
}
```

### 5.2.3. Misusing Shared Pointers

Although `shared_ptr`s improve program safety, because in general resources associated with objects are automatically released, problems are possible when objects are no longer in use. One problem just discussed is "dangling pointers" caused by cyclic dependencies.

As another problem, you have to ensure that only one group of shared pointers owns an object. The following code will not work:

Click here to view code image

```
int* p = new int;
shared_ptr<int> sp1(p);
shared_ptr<int> sp2(p);       // ERROR: two shared pointers manage allocated int
```

The problem is that both `sp1` and `sp2` would release the associated resource (call `delete`) when they lose their ownership of `p`. In general, having two owning groups means that the release of the associated resource is performed twice whenever the last owner of each group loses the ownership or gets destroyed. For this reason, you should always directly initialize a smart pointer the moment you create the object with its associated resource:

```
shared_ptr<int> sp1(new int);
shared_ptr<int> sp2(sp1);     // OK
```

This problem might also occur indirectly. In the example just introduced, suppose that we want to introduce a member function for a `Person` that creates both the reference from a kid to the parent and a corresponding reference back:

```
shared_ptr<Person> mom(new Person(name+"'s mom"));
shared_ptr<Person> dad(new Person(name+"'s dad"));
shared_ptr<Person> kid(new Person(name));
kid->setParentsAndTheirKids(mom,dad);
```

Here is a naive implementation of `setParentsAndTheirKids()`:

Click here to view code image

```
class Person {
  public:
    ...
    void setParentsAndTheirKids (shared_ptr<Person> m = nullptr,
                                 shared_ptr<Person> f = nullptr) {
        mother = m;
        father = f;
        if (m != nullptr) {
            m->kids.push_back(shared_ptr<Person>(this));   // ERROR
        }
        if (f != nullptr) {
            f->kids.push_back(shared_ptr<Person>(this));   // ERROR
        }
    }
    ...
};
```

The problem is the creation of a shared pointer out of `this`. We do that because we want to set the `kids` of members `mother` and `father`. But to do that, we need a shared pointer to the kid, which we don't have at hand. However, creating a new shared pointer out of `this` doesn't solve the issue, because we then open a new group of owners.

One way to deal with this problem is to pass the shared pointer to the kid as a third argument. But the C++ standard library provides another option: class `std::enable_shared_from_this<>` .

You can use class `std::enable_shared_from_this<>` to derive your class, representing objects managed by shared pointers, with your class name passed as template argument. Doing so allows you to use a derived member function `shared_from_this()` to create a correct `shared_ptr` out of `this` :

**Click here to view code image**

```
class Person : public std::enable_shared_from_this<Person> {
  public:
    ...
    void setParentsAndTheirKids (shared_ptr<Person> m = nullptr,
                                 shared_ptr<Person> f = nullptr) {
        mother = m;
        father = f;
        if (m != nullptr) {
            m->kids.push_back(shared_from_this());  // OK
        }
        if (f != nullptr) {
            f->kids.push_back(shared_from_this());  // OK
        }
    }
    ...
};
```

You find the whole resulting program in *util/enableshared1.cpp* .

Note that you can't call `shared_from_this()` inside the constructor (well, you can, but the result is a runtime error):

```
class Person : public std::enable_shared_from_this<Person> {
  public:
    ...
    Person (const string& n,
            shared_ptr<Person> m = nullptr,
            shared_ptr<Person> f = nullptr)
     : name(n), mother(m), father(f) {
        if (m != nullptr) {
            m->kids.push_back(shared_from_this());  // ERROR
        }
        if (f != nullptr) {
            f->kids.push_back(shared_from_this());  // ERROR
        }
    }
    ...
};
```

The problem is that `shared_ptr` stores itself in a private member of `Person` 's base class, `enable_shared_from_this<>` , *at the end* of the construction of the `Person` .

So, there is absolutely no way to create cyclic references of shared pointers during the construction of the object that initializes the shared pointer. You have to do it in two steps — one way or the other.

## 5.2.4. Shared and Weak Pointers in Detail

Let's summarize and present the whole interface that shared and weak pointers provide.

#### Class `shared_ptr` in Detail

As introduced in Section 5.2.1, page 76, class `shared_ptr` provides the concept of a smart pointer with shared ownership semantics. Whenever the last owner of a shared pointer gets destroyed, the associated object gets deleted (or the associated resources are cleaned up).

Class `shared_ptr<>` is templatized for the type of the object the initial pointer refers to:

```
namespace std {
    template <typename T>
    class shared_ptr
    {
      public:
        typedef T element_type;
        ...
    };
}
```

The element type might be `void` , which means that the shared pointer shares ownership of an object with an unspecified type, like `void*` does.

An *empty* `shared_ptr` does not share ownership of an object, so `use_count()` yields `0`. Note, however, that due to one special constructor, the shared pointer still might refer to an object.

Tables 5.3 and 5.4 list all operations provided for shared pointers.

**Table 5.3. Operations of `shared_ptr`s, Part 1**

| Operation | Effect |
|---|---|
| `shared_ptr<T> sp` | Default constructor; creates an empty shared pointer, using the default deleter (calling `delete`) |
| `shared_ptr<T> sp(ptr)` | Creates a shared pointer owning `*ptr`, using the default deleter (calling `delete`) |
| `shared_ptr<T> sp(ptr,del)` | Creates a shared pointer owning `*ptr`, using *del* as deleter |
| `shared_ptr<T> sp(ptr,del,ac)` | Creates a shared pointer owning `*ptr`, using *del* as deleter and *ac* as allocator |
| `shared_ptr<T> sp(nullptr)` | Creates an empty shared pointer, using the default deleter (calling `delete`) |
| `shared_ptr<T> sp(nullptr,del)` | Creates an empty shared pointer, using *del* as deleter |
| `shared_ptr<T> sp(nullptr, del,ac)` | Creates an empty shared pointer, using *del* as deleter and *ac* as allocator |
| `shared_ptr<T> sp(sp2)` | Creates a shared pointer sharing ownership with *sp2* |
| `shared_ptr<T> sp(move(sp2))` | Creates a shared pointer owning the pointer previously owned by *sp2* (*sp2* is empty afterward) |
| `shared_ptr<T> sp(sp2,ptr)` | Alias constructor; creates a shared pointer sharing ownership of *sp2* but referring to `*ptr` |
| `shared_ptr<T> sp(wp)` | Creates a shared pointer out of a weak pointer *wp* |
| `shared_ptr<T> sp(move(up))` | Creates a shared pointer out of a `unique_ptr` *up* |
| `shared_ptr<T> sp(move(ap))` | Creates a shared pointer out of an `auto_ptr` *ap* |
| `sp.~shared_ptr()` | Destructor; calls the deleter if *sp* owns an object |
| `sp = sp2` | Assignment (*sp* shares ownership with *sp2* afterward, giving up ownership of the object previously owned) |
| `sp = move(sp2)` | Move assignment (*sp2* transfers ownership to *sp*) |
| `sp = move(up)` | Assigns `unique_ptr` *up* (*up* transfers ownership to *sp*) |
| `sp = move(ap)` | Assigns `auto_ptr` *ap* (*ap* transfers ownership to *sp*) |
| `sp1.swap(sp2)` | Swaps pointers and deleters of *sp1* and *sp2* |
| `swap(sp1,sp2)` | Swaps pointers and deleters of *sp1* and *sp2* |
| `sp.reset()` | Gives up ownership and reinitializes the shared pointer as being empty |
| `sp.reset(ptr)` | Gives up ownership and reinitializes the shared pointer owning `*ptr`, using the default deleter (calling `delete`) |
| `sp.reset(ptr,del)` | Gives up ownership and reinitializes the shared pointer owning `*ptr`, using *del* as deleter |
| `sp.reset(ptr,del,ac)` | Gives up ownership and reinitializes the shared pointer owning `*ptr`, using *del* as deleter and *ac* as allocator |
| `make_shared(...)` | Creates a shared pointer for a new object initialized by the passed arguments |
| `allocate_shared(ac,...)` | Creates a shared pointer for a new object initialized by the passed arguments, using allocator *ac* |

**Table 5.4. Operations of `shared_ptr`s, Part 2**

| Operation | Effect |
|---|---|
| `sp.get()` | Returns the stored pointer (usually the address of the owned object or `nullptr` if none) |
| `*sp` | Returns the owned object (undefined behavior if none) |
| `sp->...` | Provides member access for the owned object (undefined behavior if none) |
| `sp.use_count()` | Returns the number of shared owners (including `sp`) or 0 if the shared pointer is empty |
| `sp.unique()` | Returns whether `sp` is the only owner (equivalent to `sp.use_count()==1` but might be faster) |
| `if (sp)` | Operator `bool()`; yields whether sp is empty |
| `sp1 == sp2` | Calls == for the stored pointers (`nullptr` is possible) |
| `sp1 != sp2` | Calls != for the stored pointers (`nullptr` is possible) |
| `sp1 < sp2` | Calls < for the stored pointers (`nullptr` is possible) |
| `sp1 <= sp2` | Calls <= for the stored pointers (`nullptr` is possible) |
| `sp1 > sp2` | Calls > for the stored pointers (`nullptr` is possible) |
| `sp1 >= sp2` | Calls >= for the stored pointers (`nullptr` is possible) |
| `static_pointer_cast(sp)` | `static_cast<>` semantic for `sp` |
| `dynamic_pointer_cast(sp)` | `dynamic_cast<>` semantic for `sp` |
| `const_pointer_cast(sp)` | `const_cast<>` semantic for `sp` |
| `get_deleter(sp)` | Returns the address of the deleter, if any, or `nullptr` otherwise |
| `strm << sp` | Calls the output operator for its raw pointer (is equal to `strm<<sp.get()`) |
| `sp.owner_before(sp2)` | Provides a strict weak ordering with another shared pointer |
| `sp.owner_before(wp)` | Provides a strict weak ordering with a weak pointer |

Whenever ownership is transferred to a shared pointer that already owned another object, the deleter for the previously owned object gets called if that shared pointer was the last owner. The same applies if a shared pointer gets a new value either by assigning a new value or by calling `reset()`: If the shared pointer previously owned an object and was the last owner, the corresponding deleter (or `delete`) gets called for the object. Note again that the passed deleter shall not throw.

The shared pointers might use different object types, provided that there is an implicit pointer conversion. For this reason, constructors, assignment operators, and `reset()` are member templates, whereas comparison operators are templatized for different types.

All comparison operators compare the raw pointers the shared pointers internally use (i.e., they call the same operator for the values returned by `get()`). They all have overloads for `nullptr` as argument. Thus, you can check whether there is a valid pointer or even whether the raw pointer is less than or greater than `nullptr`.

The constructor taking a `weak_ptr` argument throws `bad_weak_ptr` (see Section 5.2.2, page 89) if the weak pointer is empty (`expired()` yields `true`).

`get_deleter()` yields a pointer to the function defined as a deleter, if any, or `nullptr` otherwise. The pointer is valid as long as a shared pointer owns that deleter. To get the deleter, however, you have to pass its type as a template argument. For example:

```
auto del = [] (int* p) {
               delete p;
           };
std::shared_ptr<int>  p(new int, del);
decltype(del)* pd = std::get_deleter<decltype(del)>(p);
```

Note that shared pointers do not provide a `release()` operation to give up ownership and return the control of an object back to the caller. The reason is that other shared pointers might still own the object.

#### More Sophisticated `shared_ptr` Operations

A few operations are provided that might not be obvious. Most of them were motivated and introduced with [*N2351:SharedPtr*].

The constructor taking another shared pointer and an additional raw pointer is the so-called *aliasing constructor*, which allows you to capture the fact that one object owns another. For example:

```
struct X
{
   int a;
};
```

```
shared_ptr<X> px(new X);
shared_ptr<int> pi(px, &px->a);
```

The object of type X "owns" its member a, so to create a shared pointer to a, you need to keep the surrounding object alive by attaching to its reference count by means of the aliasing constructor. Other, more complex, examples exist, such as referring to a container element or to a shared library symbol.[10]

[10] Thanks to Peter Dimov for pointing this out.

Note that, as a consequence, the programmer has to ensure that the lifetimes of both objects match. Otherwise, dangling pointers or resource leaks might occur. For example:

```
shared_ptr<X> sp1(new X);
shared_ptr<X> sp2(sp1,new X);  // ERROR: delete for this X will never be called

sp1.reset();                   // deletes first X; makes sp1 empty
shared_ptr<X> sp3(sp1,new X);  // use_count()==0, but get()!=nullptr
```

Both make_shared() and allocate_shared() are provided to optimize the creation of a shared object and its associated control block (for example, maintaining the use count). Note that

```
shared_ptr<X>(new X(...))
```

performs two allocations: one for X and one for the control block used, for example, by the shared pointer to manage its use count. Using

```
make_shared<X>(...)
```

instead is considerably faster, performing only one allocation, and safer because a situation where the allocation of X succeeds but the allocation of the control block fails cannot occur. allocate_shared() allows passing your own allocator as first argument here.

The cast operators allow casting a pointer to a different type. The semantic is the same as the corresponding operators, and the result is another shared pointer of a different type. Note that using the ordinary cast operators is not possible, because it results in undefined behavior:

```
shared_ptr<void> sp(new int);  // shared pointer holds a void* internally
...
shared_ptr<int>(static_cast<int*>(sp.get()))  // ERROR: undefined behavior
static_pointer_cast<int>(sp)                  // OK
```

#### Class weak_ptr in Detail

As introduced in Section 5.2.2, page 84, class weak_ptr is a helper of class shared_ptr to share an object without owing it. Its use_count() returns the number of shared_ptr owners of an object, for which the weak_ptr s sharing the object do not count. Also, a weak_ptr can be empty, which is the case if it is not initialized by a shared_ptr or if the last owner of the corresponding object was deleted. Class weak_ptr<> is also templatized for the type of the object the initial pointer refers to:

```
namespace std {
    template <typename T>
    class weak_ptr
    {
      public:
        typedef T element_type;
        ...
    };
}
```

Table 5.5 lists all operations provided for weak pointers.

**Table 5.5. Operations of *weak_ptr*s**

| Operation | Effect |
|---|---|
| `weak_ptr<T> wp` | Default constructor; creates an empty weak pointer |
| `weak_ptr<T> wp(sp)` | Creates a weak pointer sharing ownership of the pointer owned by *sp* |
| `weak_ptr<T> wp(wp2)` | Creates a weak pointer sharing ownership of the pointer owned by *wp2* |
| `wp.~weak_ptr()` | Destructor; destroys the weak pointer but has no effect on the object owned |
| `wp = wp2` | Assignment (*wp* shares ownership of *wp2* afterward, giving up ownership of the object previously owned) |
| `wp = sp` | Assigns shared pointer *sp* (*wp* shares ownership of *sp* afterward, giving up ownership of the object previously owned) |
| `wp.swap(wp2)` | Swaps the pointers of *wp* and *wp2* |
| `swap(wp1,wp2)` | Swaps the pointers of *wp1* and *wp2* |
| `wp.reset()` | Gives up ownership of owned object, if any, and reinitializes as empty weak pointer |
| `wp.use_count()` | Returns the number of shared owners (`shared_ptr`s owning the object) or 0 if the weak pointer is empty |
| `wp.expired()` | Returns whether *wp* is empty (equivalent to `wp.use_count()==0` but might be faster) |
| `wp.lock()` | Returns a shared pointer sharing ownership of the pointer owned by the weak pointer (or an empty shared pointer if none) |
| `wp.owner_before(wp2)` | Provides a strict weak ordering with another weak pointer |
| `wp.owner_before(sp)` | Provides a strict weak ordering with a shared pointer |

The default constructor creates an empty weak pointer, which means that `expired()` yields `true`. Because `lock()` yields a shared pointer, the use count of the object increments for the lifetime of the shared pointer. This is the only way to deal with the object a weak pointer shares.

#### Thread-Safe Shared Pointer Interface

In general, shared pointers are not thread safe. Thus, to avoid undefined behavior due to data races (see Section 18.4.1, page 982), you have to use techniques, such as mutexes or locks, when shared pointers refer to the same object in multiple threads. However, reading the use count while another thread modifies it does not introduce a data race, although the value might not be up-to-date. In fact, one thread might check a use count while another thread might manipulate it. See Chapter 18 for details.

Corresponding to the atomic C-style interface for ordinary pointers (see Section 18.7.3, page 1019), overloaded versions for shared pointers are provided, which allow dealing with shared pointers concurrently. Note that concurrent access to the *pointers*, not to the values they refer to, is meant.

For example:[11]

[11] Thanks to Anthony Williams for providing this example.

```
std::shared_ptr<X> global;  // initially nullptr

void foo()
{
    std::shared_ptr<X> local{new X};
    ...
    std::atomic_store(&global,local);
}
```

Table 5.6 lists the high-level interface. A corresponding low-level interface (see Section 18.7.4, page 1019) is also provided.

**Table 5.6. High-Level Atomic Operations of `shared_ptr`**

| Operation | Effect |
|---|---|
| `atomic_is_lock_free(&sp)` | Returns `true` if the atomic interface to *sp* is lock free |
| `atomic_load(&sp)` | Returns *sp* |
| `atomic_store(&sp,sp2)` | Assigns *sp2* to *sp* |
| `atomic_exchange(&sp,sp2)` | Exchange values of *sp* and *sp2* |

### 5.2.5. Class `unique_ptr`

The `unique_ptr` type, provided by the C++ standard library since C++11, is a kind of a smart pointer that helps to avoid resource leaks when exceptions are thrown. In general, this smart pointer implements the concept of *exclusive ownership*, which means that it ensures that an object and its associated resources are "owned" only by one pointer at a time. When this owner gets destroyed or becomes empty or starts to own another object, the object previously owned also gets destroyed, and any associated resources are released.

Class `unique_ptr` succeeds class `auto_ptr`, which was originally introduced with C++98 but is deprecated now (see Section 5.2.7, page 113). Class `unique_ptr` provides a simple and clearer interface, making it less error prone than `auto_ptr`s have been.

#### Purpose of Class `unique_ptr`

Functions often operate in the following way:[12]

[12] This motivation, originally written for class `auto_ptr`, is adapted, with permission, from Scott Meyers' book *More Effective C++*. The general technique was originally presented by Bjarne Stroustrup as the "resource allocation is initialization" idiom in his books *The C++ Programming Language, 2nd edition* and *The Design and Evolution of C++*.

1. Acquire some resources.
2. Perform some operations.
3. Free the acquired resources.

If bound to local objects, the resources acquired on entry get freed automatically on function exit because the destructors of those local objects are called. But if resources are acquired explicitly and are not bound to any object, they must be freed explicitly. Resources are typically managed explicitly when pointers are used.

A typical example of using pointers in this way is the use of `new` and `delete` to create and destroy an object:

```
void f()
{
    ClassA* ptr = new ClassA;      // create an object explicitly
    ...                            // perform some operations
    delete ptr;                    // clean up (destroy the object explicitly)
}
```

This function is a source of trouble. One obvious problem is that the deletion of the object might be forgotten, especially if you have `return` statements inside the function. There also is a less obvious danger that an exception might occur. Such an exception would exit the function immediately, without calling the `delete` statement at the end of the function. The result would be a memory leak or, more generally, a resource leak.

Avoiding such a resource leak usually requires that a function catch all exceptions. For example:

```
void f()
{
    ClassA* ptr = new ClassA;      // create an object explicitly

    try {
       ...                         // perform some operations
    }
    catch (...) {                  // for any exception
        delete ptr;                // - clean up
        throw;                     // - rethrow the exception
    }

    delete ptr;                    // clean up on normal end
}
```

To handle the deletion of this object properly in the event of an exception, the code gets more complicated and redundant. If a second object is handled in this way, or if more than one `catch` clause is used, the problem gets worse. This is bad programming style and should be avoided because it is complex and error prone.

A smart pointer can help here. The smart pointer can free the data to which it points whenever the pointer itself gets destroyed. Furthermore, because it is a local variable, the pointer gets destroyed automatically when the function is exited, regardless of whether the exit is normal or is due to an exception. The class `unique_ptr` was designed to be such a smart pointer.

A `unique_ptr` is a pointer that serves as a unique *owner* of the object to which it refers. As a result, an object gets destroyed automatically when its `unique_ptr` gets destroyed. A requirement of a `unique_ptr` is that its object have only one owner.

Here is the previous example rewritten to use a `unique_ptr`:

```
// header file for unique_ptr
#include <memory>

void f()
{
    // create and initialize a unique_ptr
    std::unique_ptr<ClassA> ptr(new ClassA);
```

```
    ...                                    // perform some operations
}
```

That's all. The `delete` statement and the `catch` clause are no longer necessary.

#### Using a `unique_ptr`

A `unique_ptr` has much the same interface as an ordinary pointer; that is, operator `*` dereferences the object to which it points, whereas operator `->` provides access to a member if the object is a class or a structure:

```
// create and initialize (pointer to) string:
std::unique_ptr<std::string> up(new std::string("nico"));

(*up)[0] = 'N';                        // replace first character
up->append("lai");                     // append some characters
std::cout << *up << std::endl;         // print whole string
```

However, no pointer arithmetic, such as `++`, is defined (this counts as an advantage because pointer arithmetic is a source of trouble).

Note that class `unique_ptr<>` does not allow you to initialize an object with an ordinary pointer by using the assignment syntax. Thus, you must initialize the `unique_ptr` directly, by using its value:

```
std::unique_ptr<int> up = new int;     // ERROR
std::unique_ptr<int> up(new int);      // OK
```

A `unique_ptr` does not have to own an object, so it can be *empty*.[13] This is, for example, the case when it is initialized with the default constructor:

[13] Although the C++ standard library does define the term *empty* only for shared pointers, I don't see any reason not to do that in general.

```
std::unique_ptr<std::string> up;
```

You can also assign the `nullptr` or call `reset()`:

```
up = nullptr;
up.reset();
```

In addition, you can call `release()`, which yields the object a `unique_ptr` owned, and gives up ownership so that the caller is responsible for its object now:

```
std::unique_ptr<std::string> up(new std::string("nico"));
...
std::string* sp = up.release();  // up loses ownership
```

You can check whether a unique pointer owns an object by calling operator `bool()`:

```
if (up) {     // if up is not empty
    std::cout << *up << std::endl;
}
```

Instead, you can also compare the unique pointer with `nullptr` or query the raw pointer inside the `unique_ptr`, which yields `nullptr` if the `unique_ptr` doesn't own any object:

```
if (up != nullptr)          // if up is not empty
if (up.get() != nullptr)    // if up is not empty
```

#### Transfer of Ownership by `unique_ptr`

A `unique_ptr` provides the semantics of exclusive ownership. However, it's up to the programmer to ensure that no two unique pointers are initialized by the same pointer:

```
std::string* sp = new std::string("hello");
std::unique_ptr<std::string> up1(sp);
std::unique_ptr<std::string> up2(sp);  // ERROR: up1 and up2 own same data
```

Unfortunately, this is a runtime error, so the programmer has to avoid such a mistake.

This leads to the question of how the copy constructor and the assignment operator of `unique_ptr`s operate. The answer is simple: You can't copy or assign a unique pointer if you use the ordinary copy semantics. However, you can use the move semantics provided since C++11 (see Section 3.1.5, page 19). In that case, the constructor or assignment operator *transfers* the ownership to another unique pointer.[14]

[14] Here is the major difference with `auto_ptr`, which did transfer the ownership with the ordinary copy semantics, resulting in a source of trouble and confusion.

Consider, for example, the following use of the copy constructor:

```
// initialize a unique_ptr with a new object
std::unique_ptr<ClassA> up1(new ClassA);

// copy the unique_ptr
std::unique_ptr<ClassA> up2(up1);   // ERROR: not possible

// transfer ownership of the unique_ptr
std::unique_ptr<ClassA> up3(std::move(up1));   // OK
```

After the first statement, `up1` owns the object that was created with the `new` operator. The second, which tries to call the copy constructor, is a compile-time error because `up2` can't become another owner of that object. Only one owner at a time is allowed. However, with the third statement, we transfer ownership from `up1` to `up3`. So afterward, `up3` owns the object created with `new`, and `up1` no longer owns the object. The object created by `new ClassA` gets deleted exactly once: when `up3` gets destroyed.

The assignment operator behaves similarly:

```
// initialize a unique_ptr with a new object
std::unique_ptr<ClassA> up1(new ClassA);
std::unique_ptr<ClassA> up2;   // create another unique_ptr

up2 = up1;                     // ERROR: not possible

up2 = std::move(up1);          // assign the unique_ptr
                               // - transfers ownership from up1 to up2
```

Here, the move assignment transfers ownership from `up1` to `up2`. As a result, `up2` owns the object previously owned by `up1`.

If `up2` owned an object before an assignment, `delete` is called for that object:

**Click here to view code image**

```
// initialize a unique_ptr with a new object
std::unique_ptr<ClassA> up1(new ClassA);
// initialize another unique_ptr with a new object
std::unique_ptr<ClassA> up2(new ClassA);

up2 = std::move(up1);          // move assign the unique_ptr
                               // - delete object owned by up2
                               // - transfer ownership from up1 to up2
```

A `unique_ptr` that loses the ownership of an object without getting a new ownership refers to no object.

To assign a new value to a `unique_ptr`, this new value must also be a `unique_ptr`. You can't assign an ordinary pointer:

**Click here to view code image**

```
std::unique_ptr<ClassA> ptr;                 // create a unique_ptr

ptr = new ClassA;                            // ERROR
ptr = std::unique_ptr<ClassA>(new ClassA);   // OK, delete object owned by ptr
                                             //     and become owner of new
object
```

Assigning `nullptr` is also possible, which has the same effect as calling `reset()`:

```
up = nullptr;  // deletes the associated object, if any
```

**Source and Sink**

The transfer of ownership implies a special use for `unique_ptr`s; that is, functions can use them to transfer ownership to other functions. This can occur in two ways:

1. A function can behave as a *sink* of data. This happens if a `unique_ptr` is passed as an argument to the function by rvalue reference created with `std::move()`. In this case, the parameter of the called function gets ownership of the `unique_ptr`. Thus, if the function does not transfer it again, the object gets deleted on function exit:

**Click here to view code image**

```
void sink(std::unique_ptr<ClassA> up)      // sink() gets ownership
{
    ...
}

std::unique_ptr<ClassA> up(new ClassA);
...
sink(std::move(up));      // up loses ownership
...
```

2. A function can behave as a *source* of data. When a `unique_ptr` is returned, ownership of the returned value gets transferred to the calling context. The following example shows this technique:[15]

[15] If you assume to declare the return type as rvalue reference, don't do that; doing so would return a dangling pointer (see Section 3.1.5, page 22).

**Click here to view code image**

```
std::unique_ptr<ClassA> source()
{
    std::unique_ptr<ClassA> ptr(new ClassA);  // ptr owns the new object
    ...
    return ptr;                 // transfer ownership to calling function
}

void g()
{
    std::unique_ptr<ClassA> p;

    for (int i=0; i<10; ++i) {
        p = source();    // p gets ownership of the returned object
                         // (object previously returned by source() gets deleted)
        ...
    }
}  // last-owned object of p gets deleted
```

Each time `source()` is called, it creates an object with `new` and returns the object, along with its ownership, to the caller. The assignment of the return value to `p` transfers ownership to `p`. In the second and additional passes through the loop, the assignment to `p` deletes the object that `p` owned previously. Leaving `g()`, and thus destroying `p`, results in the destruction of the last object owned by `p`. In any case, no resource leak is possible. Even if an exception is thrown, any `unique_ptr` that owns data ensures that this data is deleted.

The reason that no `std::move()` is necessary in the return statement of `source()` is that according to the language rules of C++11, the compiler will try a move automatically (see Section 3.1.5, page 22).

#### `unique_ptr`s as Members

By using `unique_ptr`s within a class, you can also avoid resource leaks. If you use a `unique_ptr` instead of an ordinary pointer, you no longer need a destructor because the object gets deleted with the deletion of the member. In addition, a `unique_ptr` helps to avoid resource leaks caused by exceptions thrown during the initialization of an object. Note that destructors are called only if any construction is completed. So, if an exception occurs inside a constructor, destructors are called only for member objects that have been fully constructed. This can result in resource leaks for classes with multiple raw pointers if during the construction the first `new` was successful but the second was not. For example:

**Click here to view code image**

```
class ClassB {
  private:
    ClassA* ptr1;       // pointer members
    ClassA* ptr2;
  public:
    // constructor that initializes the pointers
    // - will cause resource leak if second new throws
    ClassB (int val1, int val2)
     : ptr1(new ClassA(val1)), ptr2(new ClassA(val2)) {
    }

    // copy constructor
    // - will cause resource leak if second new throws
    ClassB (const ClassB& x)
     : ptr1(new ClassA(*x.ptr1)), ptr2(new ClassA(*x.ptr2)) {
    }

    // assignment operator
```

```
    const ClassB& operator= (const ClassB& x) {
       *ptr1 = *x.ptr1;
       *ptr2 = *x.ptr2;
       return *this;
    }

    ~ClassB () {
        delete ptr1;
        delete ptr2;
    }
    ...
};
```

To avoid such a possible resource leak, you can simply use `unique_ptr` s:

**Click here to view code image**

```
class ClassB {
  private:
    std::unique_ptr<ClassA> ptr1;      // unique_ptr members
    std::unique_ptr<ClassA> ptr2;
  public:
    // constructor that initializes the unique_ptrs
    // - no resource leak possible
    ClassB (int val1, int val2)
     : ptr1(new ClassA(val1)), ptr2(new ClassA(val2)) {
    }

    // copy constructor
    // - no resource leak possible
    ClassB (const ClassB& x)
     : ptr1(new ClassA(*x.ptr1)), ptr2(new ClassA(*x.ptr2)) {
    }

    // assignment operator
    const ClassB& operator= (const ClassB& x) {
       *ptr1 = *x.ptr1;
       *ptr2 = *x.ptr2;
       return *this;
    }

    // no destructor necessary
    // (default destructor lets ptr1 and ptr2 delete their objects)
    ...
};
```

Note, first, that you can skip the destructor now because `unique_ptr` does the job for you. You also have to implement the copy constructor and assignment operator. By default, both would try to copy or assign the members, which isn't possible. If you don't provide them, `ClassB` also would provide only move semantics.

#### Dealing with Arrays

By default, `unique_ptr` s call `delete` for an object they own if they lose ownership (because they are destroyed, get a new object assigned, or become empty). Unfortunately, due to the language rules derived from C, C++ can't differentiate between the type of a pointer to one object and an array of objects. However, according to language rules for arrays, operator `delete[]` rather than `delete` has to get called. So, the following is possible but wrong:

```
std::unique_ptr<std::string> up(new std::string[10]);  // runtime ERROR
```

Now, you might assume that as for class `shared_ptr` (see Section 5.2.1, page 80), you have to define your own deleter to deal with arrays. But this is not necessary.

Fortunately, the C++ standard library provides a partial specialization of class `unique_ptr` for arrays, which calls `delete[]` for the referenced object when the pointer loses the ownership to it. So, you simply have to declare:

```
std::unique_ptr<std::string[]> up(new std::string[10]);  // OK
```

Note, however, that this partial specialization offers a slightly different interface. Instead of operators `*` and `->`, operator `[]` is provided to access one of the objects inside the referenced array:

**Click here to view code image**

```
std::unique_ptr<std::string[]> up(new std::string[10]);  // OK
...
std::cout << *up << std::endl;     // ERROR: * not defined for arrays
```

```
std::cout << up[0] << std::endl;  // OK
```

As usual, it's up to the programmer to ensure that the index is valid. Using an invalid index results in undefined behavior.

Note also that this class does not allow getting initialized by an array of a derived type. This reflects that fact that polymorphism does not work for plain arrays.

#### Class `default_delete<>`

Let's look a bit into the declaration of class `unique_ptr`. Conceptually, this class is declared as follows:[16]

[16] In the C++ standard library, class `unique_ptr` is actually more complicated because some template magic is used to specify the exact return type for operators `*` and `->`.

**Click here to view code image**

```
namespace std {
    // primary template:
    template <typename T, typename D = default_delete<T>>
    class unique_ptr
    {
      public:
        ...
        T& operator*() const;
        T* operator->() const noexcept;
        ...
    };

    // partial specialization for arrays:
    template<typename T, typename D>
    class unique_ptr<T[], D>
    {
      public:
        ...
        T& operator[](size_t i) const;
        ...
    }
}
```

Here, we can see that there is a special version of `unique_ptr` to deal with arrays. That version provides operator `[]` instead of operators `*` and `->` to deal with arrays rather than single objects. But both use class `std::default_delete<>` as deleter, which itself is specialized to call `delete[ ]` instead of `delete` for arrays:

```
namespace std {
    // primary template:
    template <typename T> class default_delete {
      public:
        void operator()(T* p) const;  // calls delete p
        ...
    };

    // partial specialization for arrays:
    template <typename T> class default_delete<T[]> {
      public:
        void operator()(T* p) const;  // calls delete[] p
        ...
    };
}
```

Note that default template arguments automatically also apply to partial specializations.

#### Deleters for Other Associated Resources

When the object you refer to requires something other than calling `delete` or `delete[ ]`, you have to specify your own deleter. Note, however, that the approach to defining a deleter differs slightly from that for `shared_ptr`s. You have to specify the type of the deleter as second template argument. That type can be a reference to a function, function pointer, or function object (see Section 6.10, page 233). If a function object is used, its "function call operator" `()` should be declared to take a pointer to the object.

For example, the following code prints an additional message before deleting an object with `delete`:

**Click here to view code image**

```
class ClassADeleter
{
  public:
    void operator () (ClassA* p) {
```

```
            std::cout << "call delete for ClassA object" << std::endl;
            delete p;
        }
    };
    ...

    std::unique_ptr<ClassA,ClassADeleter> up(new ClassA());
```

To specify a function or a lambda, you have to either declare the type of the deleter as `void(*)(T*)` or `std::function<void(T*)>` or use `decltype` (see Section 3.1.11, page 32). For example, to use your own deleter for an array of `int`s specified as a lambda, this looks as follows:

```
    std::unique_ptr<int,void(*)(int*)> up(new int[10],
                                          [](int* p) {
                                              ...
                                              delete[] p;
                                          });
```

or

**Click here to view code image**

```
    std::unique_ptr<int,std::function<void(int*)>> up(new int[10],
                                                      [](int* p) {
                                                          ...
                                                          delete[] p;
                                                      });
```

or

```
    auto l = [](int* p) {
                 ...
                 delete[] p;
             };
    std::unique_ptr<int,decltype(l)>> up(new int[10], l);
```

To avoid having to specify the *type* of the deleter when passing a function pointer or a lambda, you could also use an alias template, a language feature provided since C++11 (see Section 3.1.9, page 27):

**Click here to view code image**

```
    template <typename T>
    using uniquePtr = std::unique_ptr<T,void(*)(T*)>;   // alias template
    ...

    uniquePtr<int> up(new int[10], [](int* p) {         // used here
                                       ...
                                       delete[] p;
                                   });
```

This way, you would have more or less the same interface to specify deleters as for `shared_ptr`s.

Here is a complete example of using your own deleter:

**Click here to view code image**

```
    // util/uniqueptr1.cpp

    #include <iostream>
    #include <string>
    #include <memory>     // for unique_ptr
    #include <dirent.h>   // for opendir(), ...
    #include <cstring>    // for strerror()
    #include <cerrno>     // for errno
    using namespace std;

    class DirCloser
    {
      public:
        void operator () (DIR* dp) {
            if (closedir(dp) != 0) {
                std::cerr << "OOPS: closedir() failed" << std::endl;
            }
        }
    };

    int main()
    {
        // open current directory:
```

```
    unique_ptr<DIR,DirCloser> pDir(opendir("."));

    // process each directory entry:
    struct dirent *dp;
    while ((dp = readdir(pDir.get())) != nullptr) {
        string filename(dp->d_name);
        cout << "process " << filename << endl;
        ...
    }
}
```

Here, inside `main()`, we deal with the entries of the current directory, using the standard POSIX interface of `opendir()`, `readdir()`, and `closedir()`. To ensure that in any case the directory opened is closed by `closedir()`, we define a `unique_ptr`, which causes the `DirCloser` to be called whenever the handle referring to the opened directory gets destroyed. As for shared pointers, deleters for unique pointers may not throw. For this reason, we print only an error message here.

Another advantage of using a `unique_ptr` is that no copies are possible. Note that `readdir()` is not stateless, so it's a good idea to ensure that while using a handle to deal with a directory, a copy of the handle can't modify its state.

If you don't want to process the return value of `closedir()`, you could also pass `closedir()` directly as a function pointer, specifying that the deleter is a function pointer. But beware: The often recommended declaration

**Click here to view code image**

```
unique_ptr<DIR,int(*)(DIR*)> pDir(opendir("."),
                                  closedir);  // might not work
```

is not guaranteed to be portable, because `closedir` has `extern "C"` linkage, so in C++ code, this is not guaranteed to be convertible into `int(*)(DIR*)`. For portable code, you would need an intermediate type definition like this:[17]

[17] Thanks to Daniel Krügler for pointing this out.

```
extern "C" typedef int(*DIRDeleter)(DIR*);
unique_ptr<DIR, DIRDeleter> pDir(opendir("."),
                                 closedir);  // OK
```

Note that `closedir()` returns an `int`, so we have to specify `int(*)(DIR*)` as the type of the deleter. Note, however, that a call through a function pointer is an indirect call, which is harder to optimize away.

See Section 15.12.3, page 822, for another example of using your own `unique_ptr` deleter to restore a redirected output buffer.

### 5.2.6. Class `unique_ptr` in Detail

As introduced in Section 5.2.5, page 98, class `unique_ptr` provides the concept of a smart pointer with *exclusive ownership* semantics. Once a unique pointer has exclusive control, you cannot (accidentally) create a situation in which multiple pointers own the associated object. The major goal is to ensure that with the end of the pointer's lifetime, the associated object gets deleted (or its resource gets cleaned up). This especially helps to provide exception safety. In contrast to shared pointers, a minimum space and time overhead is the focus of this class.

Class `unique_ptr<>` is specialized for the type of the object the initial pointer refers to and its deleter:

**Click here to view code image**

```
namespace std {
    template <typename T, typename D = default_delete<T>>
    class unique_ptr
    {
      public:
        typedef ... pointer;  // may be D::pointer
        typedef T element_type;
        typedef D deleter_type;
        ...
    };
}
```

A partial specialization for arrays is provided (note that by language rules, it has the same default deleter, which is `default_delete<T[]>` then):

```
namespace std {
    template <typename T, typename D>
    class unique_ptr<T[], D>
    {
      public:
        typedef ... pointer;  // may be D::pointer
        typedef T element_type;
```

```
            typedef D deleter_type;
            ...
        };
    }
```

The element type `T` might be `void` so that the unique pointer owns an object with an unspecified type, like `void*` does. Note also that a type `pointer` is defined, which is not necessarily defined as `T*`. If the deleter `D` has a `pointer` typedef, this type will be used instead. In such a case, the template parameter `T` has only the effect of a type tag, because there is no member as part of class `unique_ptr<>` that depends on `T`; everything depends on `pointer`. The advantage is that a `unique_ptr` can thus hold other smart pointers.

If a `unique_ptr` is *empty*, it does not own an object, so `get()` returns the `nullptr`.

Table 5.7 lists all operations provided for unique pointers.

**Table 5.7. Operations of `unique_ptr`s**

| Operation | Effect |
|---|---|
| `unique_ptr<...> up` | Default constructor; creates an empty unique pointer, using an instance of the default/passed deleter type as deleter |
| `unique_ptr<T> up(nullptr)` | Creates an empty unique pointer, using an instance of the default/passed deleter type as deleter |
| `unique_ptr<...> up(ptr)` | Creates a unique pointer owning *ptr, using an instance of the default/passed deleter type as deleter |
| `unique_ptr<...> up(ptr,del)` | Creates a unique pointer owning *ptr, using *del* as deleter |
| `unique_ptr<T> up(move(up2))` | Creates a unique pointer owning the pointer previously owned by *up2* (*up2* is empty afterward) |
| `unique_ptr<T> up(move(ap))` | Creates a unique pointer owning the pointer previously owned by the `auto_ptr` *ap* (*ap* is empty afterward) |
| `up.~unique_ptr()` | Destructor; calls the deleter for an owned object |
| `up = move(up2)` | Move assignment (*up2* transfers ownership to *up*) |
| `up = nullptr` | Calls the deleter for an owned object and makes *up* empty (equivalent to `up.reset()`) |
| `up1.swap(up2)` | Swaps pointers and deleters of *up1* and *up2* |
| `swap(up1,up2)` | Swaps pointers and deleters of *up1* and *up2* |
| `up.reset()` | Calls the deleter for an owned object and makes *up* empty (equivalent to `up=nullptr`) |

| | |
|---|---|
| `up.reset(ptr)` | Calls the deleter for an owned object and reinitializes the shared pointer to own `*ptr` |
| `up.release()` | Gives up ownership back to the caller (returns owned object without calling the deleter) |
| `up.get()` | Returns the stored pointer (the address of the object owned or `nullptr` if none) |
| `*up` | Single objects only; returns the owned object (undefined behavior if none) |
| `up->...` | Single objects only; provides member access for the owned object (undefined behavior if none) |
| `up[idx]` | Array objects only; returns the element with index *idx* of the stored array (undefined behavior if none) |
| `if (up)` | Operator `bool()`; yields whether *up* is empty |
| `up1 == up2` | Calls == for the stored pointers (`nullptr` is possible) |
| `up1 != up2` | Calls != for the stored pointers (`nullptr` is possible) |
| `up1 < up2` | Calls < for the stored pointers (`nullptr` is possible) |
| `up1 <= up2` | Calls <= for the stored pointers (`nullptr` is possible) |
| `up1 > up2` | Calls > for the stored pointers (`nullptr` is possible) |
| `up1 >= up2` | Calls >= for the stored pointers (`nullptr` is possible) |
| `up.get_deleter()` | Returns a reference of the deleter |

The constructor taking a pointer and a deleter as arguments is overloaded for different types, so the following behavior is specified:

Click here to view code image

```
D d;                                          // instance of the deleter type
unique_ptr<int, D> p1(new int, D());  // D must be MoveConstructible
unique_ptr<int, D> p2(new int, d);    // D must be CopyConstructible
unique_ptr<int, D&> p3(new int, d);   // p3 holds a reference to d
unique_ptr<int, const D&> p4(new int, D());  // Error: rvalue deleter object
                                              // can't have reference deleter
type
```

For single objects, the move constructor and the assignment operator are member templates, so a type conversion is possible. All comparison operators are templatized for different element and deleter types.

All comparison operators compare the raw pointers the shared pointers internally use (call the same operator for the values returned by `get()`). They all have overloads for `nullptr` as argument. Thus, you can check whether there is a valid pointer or even if the raw pointer is less than or greater than `nullptr`.

The specialization for array types has the following differences compared to the single-object interface:

- Instead of operators `*` and `->`, operator `[]` is provided.
- The default deleter calls `delete[ ]` rather than just `delete`.
- Conversions between different types are not supported. Pointers to derived element types are especially not possible.

Note that the deleter interface differs from class `shared_ptr` (see Section 5.2.1, page 80, for details). However, as for shared pointers, deleters shall not throw.

### 5.2.7. Class `auto_ptr`

Unlike C++11, the C++98 standard library provided only one smart pointer class, `auto_ptr`, which is deprecated since C++11. Its goal was to provide the semantics that `unique_ptr` now does. However, class `auto_ptr` introduced a few problems:

- At the time of its design, the language had no move semantics for constructors and assignment operators. However, the goal was still to provide the semantics of ownership transfer. As a result, copy and assignment operators got a move semantic, which could cause serious trouble, especially when passing an `auto_ptr` as argument.
- There was no semantic of a *deleter*, so you could use it only to deal with single objects allocated with `new`.
- Because this was initially the only smart pointer provided by the C++ standard library, it was often misused, especially assuming that it provided the semantics of *shared ownership* as class `shared_ptr` does now.

Regarding the danger of unintended loss of ownership, consider the following example, consisting of a naive implementation of a function that prints the object to which an `auto_ptr` refers:

Click here to view code image

```
// this is a bad example
template <typename T>
void bad_print(std::auto_ptr<T> p)     // p gets ownership of passed argument
{
    // does p own an object ?
    if (p.get() == NULL) {
        std::cout << "NULL";
    }
    else {
        std::cout << *p;
    }
}               // Oops, exiting deletes the object to which p refers
```

Whenever an `auto_ptr` is passed to this implementation of `bad_print()`, the objects it owns, if any, are deleted. The reason is that the ownership of the `auto_ptr` that is passed as an argument is passed to the parameter `p`, and `p` deletes the object it owns on function exit. This is probably not the programmer's intention and would result in fatal runtime errors:

```
std::auto_ptr<int> p(new int);
*p = 42;          // change value to which p refers
bad_print(p);     // Oops, deletes the memory to which p refers
*p = 18;          // RUNTIME ERROR
```

That behavior especially applies when passing an `auto_ptr` to a container. With `unique_ptr`, such a mistake is no longer possible, because you explicitly would have to pass the argument with `std::move()`.

## 5.2.8. Final Words on Smart Pointers

As we have seen, C++11 provides two concepts of a smart pointer:

1. `shared_ptr` for shared ownership
2. `unique_ptr` for exclusive ownership

The latter replaces the old `auto_ptr` of C++98, which is deprecated now.

#### Performance Issues

You might wonder why the C++ standard library not only provides one smart pointer class with shared ownership semantics as such a class would also avoid resource leaks and be able to transfer ownership. The answer has to do with the performance impact of shared pointers.

Class `shared_ptr` is implemented with a nonintrusive (noninvasive) approach, which means that objects managed by this class don't have to fulfill a specific requirement, such as a common base class. The big advantage is that this concept can be applied to any type, including fundamental data types. The price is that a `shared_ptr` object internally needs multiple members: an ordinary pointer to the referenced object and a reference counter shared by all shared pointers that refer to the same object. Because weak pointers might refer to a shared object, you even need another counter. (Even if no more shared pointer uses an object, you need the counter until all weak pointers end referring to it; otherwise, you can't guarantee that they return a `use_count()` of `0`.)[18]

[18] Thanks to Howard Hinnant for pointing this out.

Thus, shared and weak pointers internally need additional helper objects, to which internal pointers refer, which means that a couple of specific optimizations are not possible (including empty base class optimizations, which would allow elimination of any memory overhead).

Unique pointers do not require any of this overhead. Their "smartness" is based on special constructors and special destructors and the elimination of copy semantics. With a stateless or empty deleter a unique pointer should consume the same amount of memory as a native pointer, and there should be no runtime overhead compared to using native pointers and doing the deletes manually. However, to avoid the introduction of unnecessary overhead, you should use function objects (including lambdas) for deleters to allow the best optimizations with, ideally, zero overhead.

#### Usage Issues

Smart pointers are not perfect, and you still have to know which problems they solve and which problems remain. For example, for any smart pointer class, you should never create multiple smart pointers out of the same ordinary pointer.

See Section 7.11, page 388, for an example of using shared pointers in multiple STL containers.

Classes `shared_ptr` and `unique_ptr` provide different approaches to deal with arrays and deleters. Class `unique_ptr` has a partial specialization for arrays, which provides a different interface. It is more flexible and provides less performance overhead but might require a bit more work to use it.

Finally, note that in general, smart pointers are not thread safe, although some guarantees apply. See Section 5.2.4, page 96, for details.