CHAPTER 12
DYNAMIC MEMORY

CONTENTS

The programs we've written so far have used objects that have well-defined lifetimes. Global objects are allocated at program start-up and destroyed when the program ends. Local, automatic objects are created and destroyed when the block in which they are defined is entered and exited. Local `static` objects are allocated before their first use and are destroyed when the program ends.

In addition to supporting automatic and `static` objects, C++ lets us allocate objects dynamically. Dynamically allocated objects have a lifetime that is independent of where they are created; they exist until they are explicitly freed.

Properly freeing dynamic objects turns out to be a surprisingly rich source of bugs. To make using dynamic objects safer, the library defines two smart pointer types that manage dynamically allocated objects. Smart pointers ensure that the objects to which they point are automatically freed when it is appropriate to do so.

Our programs have used only static or stack memory. Static memory is used for local `static` objects (§ 6.1.1, p. 205), for class `static` data members (§ 7.6, p. 300), and for variables defined outside any function. Stack memory is used for non`static` objects defined inside functions. Objects allocated in static or stack memory are automatically created and destroyed by the compiler. Stack objects exist only while the block in which they are defined is executing; `static` objects are allocated before they are used, and they are destroyed when the program ends.

In addition to static or stack memory, every program also has a pool of memory that it can use. This memory is referred to as the **free store** or **heap**. Programs use the heap for objects that they **dynamically allocate**—that is, for objects that the program allocates at run time. The program controls the lifetime of dynamic objects; our code must explicitly destroy such objects when they are no longer needed.

WARNING Although necessary at times, dynamic memory is notoriously tricky to manage correctly.

12.1 Dynamic Memory and Smart Pointers

In C++, dynamic memory is managed through a pair of operators: **`new`**, which allocates, and optionally initializes, an object in dynamic memory and returns a pointer to that object; and **`delete`**, which takes a pointer to a dynamic object, destroys that object, and frees the associated memory.

Dynamic memory is problematic because it is surprisingly hard to ensure that we free memory at the right time. Either we forget to free the memory—in which case we have a memory leak—or we free the memory when there are still pointers referring to that memory—in which case we have a pointer that refers to memory that is no longer valid.

C++ 11 To make using dynamic memory easier (and safer), the new library provides two **smart pointer** types that manage dynamic objects. A smart pointer acts like a regular pointer with the important exception that it automatically deletes the object to which it points. The new library defines two kinds of smart pointers that differ in how they manage their underlying pointers: **`shared_ptr`**, which allows multiple pointers to refer to the same object, and **`unique_ptr`**, which "owns" the object to which it points. The library also defines a companion class named **`weak_ptr`** that is a weak reference to an object managed by a `shared_ptr`. All three are defined in the `memory` header.

12.1.1 The `shared_ptr` Class

Like `vector`s, smart pointers are templates (§ 3.3, p. 96). Therefore, when we create a smart pointer, we must supply additional information—in this case, the type to which the pointer can point. As with `vector`, we supply that type inside angle brackets that follow the name of the kind of smart pointer we are defining:

```
shared_ptr<string> p1;     // shared_ptr that can point at a string
```

```
shared_ptr<list<int>> p2; //  shared_ptr that can point at a list of ints
```

A default initialized smart pointer holds a null pointer (§ 2.3.2, p. 53). In § 12.1.3 (p. 464), we'll cover additional ways to initialize a smart pointer.

We use a smart pointer in ways that are similar to using a pointer. Dereferencing a smart pointer returns the object to which the pointer points. When we use a smart pointer in a condition, the effect is to test whether the pointer is null:

```
// if p1 is not null, check whether it's the empty string
if (p1 && p1->empty())
    *p1 = "hi";  // if so, dereference p1 to assign a new value to that string
```

Table 12.1 (overleaf) lists operations common to `shared_ptr` and `unique_ptr`. Those that are particular to `shared_ptr` are listed in Table 12.2 (p. 453).

The `make_shared` Function

The safest way to allocate and use dynamic memory is to call a library function named `make_shared`. This function allocates and initializes an object in dynamic memory and returns a `shared_ptr` that points to that object. Like the smart pointers, `make_shared` is defined in the `memory` header.

When we call `make_shared`, we must specify the type of object we want to create. We do so in the same way as we use a template class, by following the function name with a type enclosed in angle brackets:

```
// shared_ptr that points to an int with value 42
shared_ptr<int> p3 = make_shared<int>(42);
// p4 points to a string with value 9999999999
shared_ptr<string> p4 = make_shared<string>(10, '9');
// p5 points to an int that is value initialized (§ 3.3.1 (p. 98)) to 0
shared_ptr<int> p5 = make_shared<int>();
```

Like the sequential-container `emplace` members (§ 9.3.1, p. 345), `make_shared` uses its arguments to construct an object of the given type. For example, a call to `make_shared<string>` must pass argument(s) that match one of the `string` constructors. Calls to `make_shared<int>` can pass any value we can use to initialize an `int`. And so on. If we do not pass any arguments, then the object is value initialized (§ 3.3.1, p. 98).

Of course, ordinarily we use `auto` (§ 2.5.2, p. 68) to make it easier to define an object to hold the result of `make_shared`:

```
// p6 points to a dynamically allocated, empty vector<string>
auto p6 = make_shared<vector<string>>();
```

Copying and Assigning `shared_ptr`s

When we copy or assign a `shared_ptr`, each `shared_ptr` keeps track of how many other `shared_ptr`s point to the same object:

```
auto p = make_shared<int>(42); // object to which p points has one user
auto q(p); // p and q point to the same object
           // object to which p and q point has two users
```

Table 12.1: Operations Common to `shared_ptr` and `unique_ptr`

`shared_ptr<T> sp`
`unique_ptr<T> up`
	Null smart pointer that can point to objects of type `T`.

`p`	Use `p` as a condition; `true` if `p` points to an object.
`*p`	Dereference `p` to get the object to which `p` points.
`p->mem`	Synonym for `(*p).mem`.
`p.get()`	Returns the pointer in `p`. Use with caution; the object to which the returned pointer points will disappear when the smart pointer deletes it.

`swap(p, q)`
`p.swap(q)`
	Swaps the pointers in `p` and `q`.

We can think of a `shared_ptr` as if it has an associated counter, usually referred to as a **reference count**. Whenever we copy a `shared_ptr`, the count is incremented. For example, the counter associated with a `shared_ptr` is incremented when we use it to initialize another `shared_ptr`, when we use it as the right-hand operand of an assignment, or when we pass it to (§ 6.2.1, p. 209) or return it from a function by value (§ 6.3.2, p. 224). The counter is decremented when we assign a new value to the `shared_ptr` and when the `shared_ptr` itself is destroyed, such as when a local `shared_ptr` goes out of scope (§ 6.1.1, p. 204).

Once a `shared_ptr`'s counter goes to zero, the `shared_ptr` automatically frees the object that it manages:

```
auto r = make_shared<int>(42); // int to which r points has one user
r = q;  // assign to r, making it point to a different address
        // increase the use count for the object to which q points
        // reduce the use count of the object to which r had pointed
        // the object r had pointed to has no users; that object is automatically freed
```

Here we allocate an `int` and store a pointer to that `int` in `r`. Next, we assign a new value to `r`. In this case, `r` is the only `shared_ptr` pointing to the one we previously allocated. That `int` is automatically freed as part of assigning `q` to `r`.

Note

It is up to the implementation whether to use a counter or another data structure to keep track of how many pointers share state. The key point is that the class keeps track of how many `shared_ptrs` point to the same object and automatically frees that object when appropriate.

`shared_ptrs` Automatically Destroy Their Objects ...

When the last `shared_ptr` pointing to an object is destroyed, the `shared_ptr` class automatically destroys the object to which that `shared_ptr` points. It does so through another special member function known as a **destructor**. Analogous to its constructors, each class has a destructor. Just as a constructor controls initialization, the destructor controls what happens when objects of that class type are destroyed.

Table 12.2: Operations Specific to `shared_ptr`

`make_shared<T>(`*args*`)`	Returns a `shared_ptr` pointing to a dynamically allocated object of type `T`. Uses *args* to initialize that object.
`shared_ptr<T> p(q)`	`p` is a copy of the `shared_ptr` `q`; increments the count in `q`. The pointer in `q` must be convertible to `T*` (§ 4.11.2, p. 161).
`p = q`	`p` and `q` are `shared_ptr`s holding pointers that can be converted to one another. Decrements `p`'s reference count and increments `q`'s count; deletes `p`'s existing memory if `p`'s count goes to 0.
`p.unique()`	Returns `true` if `p.use_count()` is one; `false` otherwise.
`p.use_count()`	Returns the number of objects sharing with `p`; may be a slow operation, intended primarily for debugging purposes.

Destructors generally free the resources that an object has allocated. For example, the `string` constructors (and other `string` members) allocate memory to hold the characters that compose the `string`. The `string` destructor frees that memory. Similarly, several `vector` operations allocate memory to hold the elements in the `vector`. The destructor for `vector` destroys those elements and frees the memory used for the elements.

The destructor for `shared_ptr` decrements the reference count of the object to which that `shared_ptr` points. If the count goes to zero, the `shared_ptr` destructor destroys the object to which the `shared_ptr` points and frees the memory used by that object.

. . . and Automatically Free the Associated Memory

The fact that the `shared_ptr` class automatically frees dynamic objects when they are no longer needed makes it fairly easy to use dynamic memory. For example, we might have a function that returns a `shared_ptr` to a dynamically allocated object of a type named `Foo` that can be initialized by an argument of type `T`:

```
// factory returns a shared_ptr pointing to a dynamically allocated object
shared_ptr<Foo> factory(T arg)
{
    // process arg as appropriate
    // shared_ptr will take care of deleting this memory
    return make_shared<Foo>(arg);
}
```

Because `factory` returns a `shared_ptr`, we can be sure that the object allocated by `factory` will be freed when appropriate. For example, the following function stores the `shared_ptr` returned by `factory` in a local variable:

```
void use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);
    // use p
} // p goes out of scope; the memory to which p points is automatically freed
```

Because p is local to use_factory, it is destroyed when use_factory ends (§ 6.1.1, p. 204). When p is destroyed, its reference count is decremented and checked. In this case, p is the only object referring to the memory returned by factory. Because p is about to go away, the object to which p points will be destroyed and the memory in which that object resides will be freed.

The memory will not be freed if there is any other shared_ptr pointing to it:

```
shared_ptr<Foo> use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);
    // use p
    return p;   // reference count is incremented when we return p
} // p goes out of scope; the memory to which p points is not freed
```

In this version, the return statement in use_factory returns a copy of p to its caller (§ 6.3.2, p. 224). Copying a shared_ptr adds to the reference count of that object. Now when p is destroyed, there will be another user for the memory to which p points. The shared_ptr class ensures that so long as there are any shared_ptrs attached to that memory, the memory itself will not be freed.

Because memory is not freed until the last shared_ptr goes away, it can be important to be sure that shared_ptrs don't stay around after they are no longer needed. The program will execute correctly but may waste memory if you neglect to destroy shared_ptrs that the program does not need. One way that shared_ptrs might stay around after you need them is if you put shared_ptrs in a container and subsequently reorder the container so that you don't need all the elements. You should be sure to erase shared_ptr elements once you no longer need those elements.

Note

If you put shared_ptrs in a container, and you subsequently need to use some, but not all, of the elements, remember to erase the elements you no longer need.

Classes with Resources That Have Dynamic Lifetime

Programs tend to use dynamic memory for one of three purposes:

1. They don't know how many objects they'll need
2. They don't know the precise type of the objects they need
3. They want to share data between several objects

The container classes are an example of classes that use dynamic memory for the first purpose and we'll see examples of the second in Chapter 15. In this section, we'll define a class that uses dynamic memory in order to let several objects share the same underlying data.

So far, the classes we've used allocate resources that exist only as long as the corresponding objects. For example, each vector "owns" its own elements. When we copy a vector, the elements in the original vector and in the copy are separate from one another:

```
vector<string> v1;  // empty vector
{ // new scope
    vector<string> v2 = {"a", "an", "the"};
    v1 = v2; // copies the elements from v2 into v1
} // v2 is destroyed, which destroys the elements in v2
  // v1 has three elements, which are copies of the ones originally in v2
```

The elements allocated by a `vector` exist only while the `vector` itself exists. When a `vector` is destroyed, the elements in the `vector` are also destroyed.

Some classes allocate resources with a lifetime that is independent of the original object. As an example, assume we want to define a class named `Blob` that will hold a collection of elements. Unlike the containers, we want `Blob` objects that are copies of one another to share the same elements. That is, when we copy a `Blob`, the original and the copy should refer to the same underlying elements.

In general, when two objects share the same underlying data, we can't unilaterally destroy the data when an object of that type goes away:

```
Blob<string> b1;    // empty Blob
{ // new scope
    Blob<string> b2 = {"a", "an", "the"};
    b1 = b2; // b1 and b2 share the same elements
} // b2 is destroyed, but the elements in b2 must not be destroyed
  // b1 points to the elements originally created in b2
```

In this example, `b1` and `b2` share the same elements. When `b2` goes out of scope, those elements must stay around, because `b1` is still using them.

Note: One common reason to use dynamic memory is to allow multiple objects to share the same state.

Defining the `StrBlob` Class

Ultimately, we'll implement our `Blob` class as a template, but we won't learn how to do so until § 16.1.2 (p. 658). For now, we'll define a version of our class that can manage `string`s. As a result, we'll name this version of our class `StrBlob`.

The easiest way to implement a new collection type is to use one of the library containers to manage the elements. That way, we can let the library type manage the storage for the elements themselves. In this case, we'll use a `vector` to hold our elements.

However, we can't store the `vector` directly in a `Blob` object. Members of an object are destroyed when the object itself is destroyed. For example, assume that `b1` and `b2` are two `Blob`s that share the same `vector`. If that `vector` were stored in one of those `Blob`s—say, `b2`—then that `vector`, and therefore its elements, would no longer exist once `b2` goes out of scope. To ensure that the elements continue to exist, we'll store the `vector` in dynamic memory.

To implement the sharing we want, we'll give each `StrBlob` a `shared_ptr` to a dynamically allocated `vector`. That `shared_ptr` member will keep track of how many `StrBlob`s share the same `vector` and will delete the `vector` when the last `StrBlob` using that `vector` is destroyed.

We still need to decide what operations our class will provide. For now, we'll implement a small subset of the vector operations. We'll also change the operations that access elements (e.g., front and back): In our class, these operations will throw an exception if a user attempts to access an element that doesn't exist.

Our class will have a default constructor and a constructor that has a parameter of type initializer_list<string> (§ 6.2.6, p. 220). This constructor will take a braced list of initializers.

```
class StrBlob {
public:
    typedef std::vector<std::string>::size_type size_type;
    StrBlob();
    StrBlob(std::initializer_list<std::string> il);
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
    // add and remove elements
    void push_back(const std::string &t) {data->push_back(t);}
    void pop_back();
    // element access
    std::string& front();
    std::string& back();
private:
    std::shared_ptr<std::vector<std::string>> data;
    // throws msg if data[i] isn't valid
    void check(size_type i, const std::string &msg) const;
};
```

Inside the class we implemented the size, empty, and push_back members. These members forward their work through the data pointer to the underlying vector. For example, size() on a StrBlob calls data->size(), and so on.

StrBlob Constructors

Each constructor uses its constructor initializer list (§ 7.1.4, p. 265) to initialize its data member to point to a dynamically allocated vector. The default constructor allocates an empty vector:

```
StrBlob::StrBlob(): data(make_shared<vector<string>>()) { }
StrBlob::StrBlob(initializer_list<string> il):
              data(make_shared<vector<string>>(il)) { }
```

The constructor that takes an initializer_list passes its parameter to the corresponding vector constructor (§ 2.2.1, p. 43). That constructor initializes the vector's elements by copying the values in the list.

Element Access Members

The pop_back, front, and back operations access members in the vector. These operations must check that an element exists before attempting to access that element. Because several members need to do the same checking, we've given our class a `private` utility function named `check` that verifies that a given index is in range. In addition to an index, `check` takes a `string` argument that it will pass to the exception handler. The `string` describes what went wrong:

```
void StrBlob::check(size_type i, const string &msg) const
{
    if (i >= data->size())
        throw out_of_range(msg);
}
```

The `pop_back` and element access members first call `check`. If `check` succeeds, these members forward their work to the underlying `vector` operation:

```
string& StrBlob::front()
{
    // if the vector is empty, check will throw
    check(0, "front on empty StrBlob");
    return data->front();
}
string& StrBlob::back()
{
    check(0, "back on empty StrBlob");
    return data->back();
}
void StrBlob::pop_back()
{
    check(0, "pop_back on empty StrBlob");
    data->pop_back();
}
```

The `front` and `back` members should be overloaded on `const` (§ 7.3.2, p. 276). Defining those versions is left as an exercise.

Copying, Assigning, and Destroying `StrBlob`s

Like our `Sales_data` class, `StrBlob` uses the default versions of the operations that copy, assign, and destroy objects of its type (§ 7.1.5, p. 267). By default, these operations copy, assign, and destroy the data members of the class. Our `StrBlob` has only one data member, which is a `shared_ptr`. Therefore, when we copy, assign, or destroy a `StrBlob`, its `shared_ptr` member will be copied, assigned, or destroyed.

As we've seen, copying a `shared_ptr` increments its reference count; assigning one `shared_ptr` to another increments the count of the right-hand operand and decrements the count in the left-hand operand; and destroying a `shared_ptr` decrements the count. If the count in a `shared_ptr` goes to zero, the object to which that `shared_ptr` points is automatically destroyed. Thus, the `vector` allocated by the `StrBlob` constructors will be automatically destroyed when the last `StrBlob` pointing to that `vector` is destroyed.

Exercises Section 12.1.1

Exercise 12.1: How many elements do `b1` and `b2` have at the end of this code?

```
StrBlob b1;
{
    StrBlob b2 = {"a", "an", "the"};
    b1 = b2;
    b2.push_back("about");
}
```

Exercise 12.2: Write your own version of the `StrBlob` class including the `const` versions of `front` and `back`.

Exercise 12.3: Does this class need `const` versions of `push_back` and `pop_back`? If so, add them. If not, why aren't they needed?

Exercise 12.4: In our `check` function we didn't check whether `i` was greater than zero. Why is it okay to omit that check?

Exercise 12.5: We did not make the constructor that takes an `initializer_list` `explicit` (§ 7.5.4, p. 296). Discuss the pros and cons of this design choice.

12.1.2 Managing Memory Directly

The language itself defines two operators that allocate and free dynamic memory. The `new` operator allocates memory, and `delete` frees memory allocated by `new`.

For reasons that will become clear as we describe how these operators work, using these operators to manage memory is considerably more error-prone than using a smart pointer. Moreover, classes that do manage their own memory—unlike those that use smart pointers—cannot rely on the default definitions for the members that copy, assign, and destroy class objects (§ 7.1.4, p. 264). As a result, programs that use smart pointers are likely to be easier to write and debug.

WARNING

Until you have read Chapter 13, your classes should allocate dynamic memory *only* if they use smart pointers to manage that memory.

Using `new` to Dynamically Allocate and Initialize Objects

Objects allocated on the free store are unnamed, so **`new`** offers no way to name the objects that it allocates. Instead, `new` returns a pointer to the object it allocates:

```
int *pi = new int;        // pi points to a dynamically allocated,
                          // unnamed, uninitialized int
```

This `new` expression constructs an object of type `int` on the free store and returns a pointer to that object.

By default, dynamically allocated objects are default initialized (§ 2.2.1, p. 43), which means that objects of built-in or compound type have undefined value; objects of class type are initialized by their default constructor:

```
string *ps = new string;   // initialized to empty string
int *pi = new int;         // pi points to an uninitialized int
```

We can initialize a dynamically allocated object using direct initialization (§ 3.2.1, p. 84). We can use traditional construction (using parentheses), and under the new standard, we can also use list initialization (with curly braces):

C++ 11

```
int *pi = new int(1024); // object to which pi points has value 1024
string *ps = new string(10, '9');   // *ps is "9999999999"
// vector with ten elements with values from 0 to 9
vector<int> *pv = new vector<int>{0,1,2,3,4,5,6,7,8,9};
```

We can also value initialize (§ 3.3.1, p. 98) a dynamically allocated object by following the type name with a pair of empty parentheses:

```
string *ps1 = new string;  // default initialized to the empty string
string *ps = new string(); // value initialized to the empty string
int *pi1 = new int;        // default initialized; *pi1 is undefined
int *pi2 = new int();      // value initialized to 0; *pi2 is 0
```

For class types (such as `string`) that define their own constructors (§ 7.1.4, p. 262), requesting value initialization is of no consequence; regardless of form, the object is initialized by the default constructor. In the case of built-in types the difference is significant; a value-initialized object of built-in type has a well-defined value but a default-initialized object does not. Similarly, members of built-in type in classes that rely on the synthesized default constructor will also be uninitialized if those members are not initialized in the class body (§ 7.1.4, p. 263).

Best Practices

For the same reasons as we usually initialize variables, it is also a good idea to initialize dynamically allocated objects.

When we provide an initializer inside parentheses, we can use `auto` (§ 2.5.2, p. 68) to deduce the type of the object we want to allocate from that initializer. However, because the compiler uses the initializer's type to deduce the type to allocate, we can use `auto` only with a single initializer inside parentheses:

C++ 11

```
auto p1 = new auto(obj);    // p points to an object of the type of obj
                            // that object is initialized from obj
auto p2 = new auto{a,b,c}; // error: must use parentheses for the initializer
```

The type of `p1` is a pointer to the `auto`-deduced type of `obj`. If `obj` is an `int`, then `p1` is `int*`; if `obj` is a `string`, then `p1` is a `string*`; and so on. The newly allocated object is initialized from the value of `obj`.

Dynamically Allocated `const` Objects

It is legal to use `new` to allocate `const` objects:

```
// allocate and initialize a const int
const int *pci = new const int(1024);
// allocate a default-initialized const empty string
const string *pcs = new const string;
```

Like any other `const`, a dynamically allocated `const` object must be initialized. A `const` dynamic object of a class type that defines a default constructor (§ 7.1.4, p. 263) may be initialized implicitly. Objects of other types must be explicitly initialized. Because the allocated object is `const`, the pointer returned by `new` is a pointer to `const` (§ 2.4.2, p. 62).

Memory Exhaustion

Although modern machines tend to have huge memory capacity, it is always possible that the free store will be exhausted. Once a program has used all of its available memory, `new` expressions will fail. By default, if `new` is unable to allocate the requested storage, it throws an exception of type `bad_alloc` (§ 5.6, p. 193). We can prevent `new` from throwing an exception by using a different form of `new`:

```
// if allocation fails, new returns a null pointer
int *p1 = new int; // if allocation fails, new throws std::bad_alloc
int *p2 = new (nothrow) int; // if allocation fails, new returns a null pointer
```

For reasons we'll explain in § 19.1.2 (p. 824) this form of `new` is referred to as **placement new**. A placement new expression lets us pass additional arguments to `new`. In this case, we pass an object named `nothrow` that is defined by the library. When we pass `nothrow` to `new`, we tell `new` that it must not throw an exception. If this form of `new` is unable to allocate the requested storage, it will return a null pointer. Both `bad_alloc` and `nothrow` are defined in the `new` header.

Freeing Dynamic Memory

In order to prevent memory exhaustion, we must return dynamically allocated memory to the system once we are finished using it. We return memory through a **`delete` expression**. A `delete` expression takes a pointer to the object we want to free:

```
delete p;        // p must point to a dynamically allocated object or be null
```

Like `new`, a `delete` expression performs two actions: It destroys the object to which its given pointer points, and it frees the corresponding memory.

Pointer Values and `delete`

The pointer we pass to `delete` must either point to dynamically allocated memory or be a null pointer (§ 2.3.2, p. 53). Deleting a pointer to memory that was not allocated by `new`, or deleting the same pointer value more than once, is undefined:

```
int i, *pi1 = &i, *pi2 = nullptr;
double *pd = new double(33), *pd2 = pd;
delete i;    // error: i is not a pointer
delete pi1; // undefined: pi1 refers to a local
delete pd;  // ok
delete pd2; // undefined: the memory pointed to by pd2 was already freed
delete pi2; // ok: it is always ok to delete a null pointer
```

The compiler will generate an error for the `delete` of `i` because it knows that `i` is not a pointer. The errors associated with executing `delete` on `pi1` and `pd2` are more insidious: In general, compilers cannot tell whether a pointer points to a statically or dynamically allocated object. Similarly, the compiler cannot tell whether memory addressed by a pointer has already been freed. Most compilers will accept these `delete` expressions, even though they are in error.

Although the value of a `const` object cannot be modified, the object itself can be destroyed. As with any other dynamic object, a `const` dynamic object is freed by executing `delete` on a pointer that points to that object:

```
const int *pci = new const int(1024);
delete pci;  // ok: deletes a const object
```

Dynamically Allocated Objects Exist until They Are Freed

As we saw in § 12.1.1 (p. 452), memory that is managed through a `shared_ptr` is automatically deleted when the last `shared_ptr` is destroyed. The same is not true for memory we manage using built-in pointers. A dynamic object managed through a built-in pointer exists until it is explicitly deleted.

Functions that return pointers (rather than smart pointers) to dynamic memory put a burden on their callers—the caller must remember to delete the memory:

```
// factory returns a pointer to a dynamically allocated object
Foo* factory(T arg)
{
    // process arg as appropriate
    return new Foo(arg); // caller is responsible for deleting this memory
}
```

Like our earlier `factory` function (§ 12.1.1, p. 453), this version of `factory` allocates an object but does not `delete` it. Callers of `factory` are responsible for freeing this memory when they no longer need the allocated object. Unfortunately, all too often the caller forgets to do so:

```
void use_factory(T arg)
{
    Foo *p = factory(arg);
    // use p but do not delete it
} // p goes out of scope, but the memory to which p points is not freed!
```

Here, our `use_factory` function calls `factory`, which allocates a new object of type `Foo`. When `use_factory` returns, the local variable `p` is destroyed. That variable is a built-in pointer, not a smart pointer.

Unlike class types, nothing happens when objects of built-in type are destroyed. In particular, when a pointer goes out of scope, nothing happens to the object to which the pointer points. If that pointer points to dynamic memory, that memory is not automatically freed.

WARNING

Dynamic memory managed through built-in pointers (rather than smart pointers) exists until it is explicitly freed.

In this example, `p` was the only pointer to the memory allocated by `factory`. Once `use_factory` returns, the program has no way to free that memory. Depending on the logic of our overall program, we should fix this bug by remembering to free the memory inside `use_factory`:

```
void use_factory(T arg)
{
    Foo *p = factory(arg);
    // use p
    delete p;  // remember to free the memory now that we no longer need it
}
```

or, if other code in our system needs to use the object allocated by `use_factory`, we should change that function to return a pointer to the memory it allocated:

```
Foo* use_factory(T arg)
{
    Foo *p = factory(arg);
    // use p
    return p;  // caller must delete the memory
}
```

CAUTION: MANAGING DYNAMIC MEMORY IS ERROR-PRONE

There are three common problems with using `new` and `delete` to manage dynamic memory:

1. **Forgetting to `delete` memory. Neglecting to delete dynamic memory is known as a "memory leak," because the memory is never returned to the free store. Testing for memory leaks is difficult because they usually cannot be detected until the application is run for a long enough time to actually exhaust memory.**
2. **Using an object after it has been deleted. This error can sometimes be detected by making the pointer null after the delete.**
3. **Deleting the same memory twice. This error can happen when two pointers address the same dynamically allocated object. If `delete` is applied to one of the pointers, then the object's memory is returned to the free store. If we subsequently `delete` the second pointer, then the free store may be corrupted.**

These kinds of errors are considerably easier to make than they are to find and fix.

Best Practices

You can avoid *all* of these problems by using smart pointers exclusively. The smart pointer will take care of deleting the memory *only* when there are no remaining smart pointers pointing to that memory.

Resetting the Value of a Pointer after a `delete` . . .

When we `delete` a pointer, that pointer becomes invalid. Although the pointer is invalid, on many machines the pointer continues to hold the address of the (freed) dynamic memory. After the `delete`, the pointer becomes what is referred to as a

dangling pointer. A dangling pointer is one that refers to memory that once held an object but no longer does so.

Dangling pointers have all the problems of uninitialized pointers (§ 2.3.2, p. 54). We can avoid the problems with dangling pointers by deleting the memory associated with a pointer just before the pointer itself goes out of scope. That way there is no chance to use the pointer after the memory associated with the pointer is freed. If we need to keep the pointer around, we can assign `nullptr` to the pointer after we use `delete`. Doing so makes it clear that the pointer points to no object.

. . . Provides Only Limited Protection

A fundamental problem with dynamic memory is that there can be several pointers that point to the same memory. Resetting the pointer we use to `delete` that memory lets us check that particular pointer but has no effect on any of the other pointers that still point at the (freed) memory. For example:

```
int *p(new int(42));   //  p points to dynamic memory
auto q = p;            //  p and q point to the same memory
delete p;     //  invalidates both p and q
p = nullptr;  //  indicates that p is no longer bound to an object
```

Here both `p` and `q` point at the same dynamically allocated object. We `delete` that memory and set `p` to `nullptr`, indicating that the pointer no longer points to an object. However, resetting `p` has no effect on `q`, which became invalid when we deleted the memory to which `p` (and `q`!) pointed. In real systems, finding all the pointers that point to the same memory is surprisingly difficult.

EXERCISES SECTION 12.1.2

Exercise 12.6: Write a function that returns a dynamically allocated `vector` of `int`s. Pass that `vector` to another function that reads the standard input to give values to the elements. Pass the `vector` to another function to print the values that were read. Remember to `delete` the `vector` at the appropriate time.

Exercise 12.7: Redo the previous exercise, this time using `shared_ptr`.

Exercise 12.8: Explain what if anything is wrong with the following function.

```
bool b() {
    int* p = new int;
    // ...
    return p;
}
```

Exercise 12.9: Explain what happens in the following code:

```
int *q = new int(42), *r = new int(100);
r = q;
auto q2 = make_shared<int>(42), r2 = make_shared<int>(100);
r2 = q2;
```

12.1.3 Using `shared_ptrs` with `new`

As we've seen, if we do not initialize a smart pointer, it is initialized as a null pointer. As described in Table 12.3, we can also initialize a smart pointer from a pointer returned by `new`:

```
shared_ptr<double> p1; // shared_ptr that can point at a double
shared_ptr<int> p2(new int(42)); // p2 points to an int with value 42
```

The smart pointer constructors that take pointers are `explicit` (§ 7.5.4, p. 296). Hence, we cannot implicitly convert a built-in pointer to a smart pointer; we must use the direct form of initialization (§ 3.2.1, p. 84) to initialize a smart pointer:

```
shared_ptr<int> p1 = new int(1024); // error: must use direct initialization
shared_ptr<int> p2(new int(1024));  // ok: uses direct initialization
```

The initialization of `p1` implicitly asks the compiler to create a `shared_ptr` from the `int*` returned by `new`. Because we can't implicitly convert a pointer to a smart pointer, this initialization is an error. For the same reason, a function that returns a `shared_ptr` cannot implicitly convert a plain pointer in its return statement:

```
shared_ptr<int> clone(int p) {
    return new int(p); // error: implicit conversion to shared_ptr<int>
}
```

We must explicitly bind a `shared_ptr` to the pointer we want to return:

```
shared_ptr<int> clone(int p) {
    // ok: explicitly create a shared_ptr<int> from int*
    return shared_ptr<int>(new int(p));
}
```

By default, a pointer used to initialize a smart pointer must point to dynamic memory because, by default, smart pointers use `delete` to free the associated object. We can bind smart pointers to pointers to other kinds of resources. However, to do so, we must supply our own operation to use in place of `delete`. We'll see how to supply our own deletion code in § 12.1.4 (p. 468).

Don't Mix Ordinary Pointers and Smart Pointers ...

A `shared_ptr` can coordinate destruction only with other `shared_ptrs` that are copies of itself. Indeed, this fact is one of the reasons we recommend using `make_shared` rather than `new`. That way, we bind a `shared_ptr` to the object at the same time that we allocate it. There is no way to inadvertently bind the same memory to more than one independently created `shared_ptr`.

Consider the following function that operates on a `shared_ptr`:

```
// ptr is created and initialized when process is called
void process(shared_ptr<int> ptr)
{
    // use ptr
} // ptr goes out of scope and is destroyed
```

Table 12.3: Other Ways to Define and Change `shared_ptrs`

`shared_ptr<T> p(q)`	`p` manages the object to which the built-in pointer `q` points; `q` must point to memory allocated by `new` and must be convertible to `T*`.
`shared_ptr<T> p(u)`	`p` assumes ownership from the `unique_ptr` `u`; makes `u` null.
`shared_ptr<T> p(q, d)`	`p` assumes ownership for the object to which the built-in pointer `q` points. `q` must be convertible to `T*` (§ 4.11.2, p. 161). `p` will use the callable object `d` (§ 10.3.2, p. 388) in place of `delete` to free `q`.
`shared_ptr<T> p(p2, d)`	`p` is a copy of the `shared_ptr` `p2` as described in Table 12.2 except that `p` uses the callable object `d` in place of `delete`.

`p.reset()`
`p.reset(q)`
`p.reset(q, d)`
	If `p` is the only `shared_ptr` pointing at its object, `reset` frees `p`'s existing object. If the optional built-in pointer `q` is passed, makes `p` point to `q`, otherwise makes `p` null. If `d` is supplied, will call `d` to free `q` otherwise uses `delete` to free `q`.

The parameter to `process` is passed by value, so the argument to `process` is copied into `ptr`. Copying a `shared_ptr` increments its reference count. Thus, inside `process` the count is at least 2. When `process` completes, the reference count of `ptr` is decremented but cannot go to zero. Therefore, when the local variable `ptr` is destroyed, the memory to which `ptr` points will not be deleted.

The right way to use this function is to pass it a `shared_ptr`:

```
shared_ptr<int> p(new int(42)); // reference count is 1
process(p); // copying p increments its count; in process the reference count is 2
int i = *p; // ok: reference count is 1
```

Although we cannot pass a built-in pointer to `process`, we can pass `process` a (temporary) `shared_ptr` that we explicitly construct from a built-in pointer. However, doing so is likely to be an error:

```
int *x(new int(1024)); // dangerous: x is a plain pointer, not a smart pointer
process(x);  // error: cannot convert int* to shared_ptr<int>
process(shared_ptr<int>(x)); // legal, but the memory will be deleted!
int j = *x;  // undefined: x is a dangling pointer!
```

In this call, we passed a temporary `shared_ptr` to `process`. That temporary is destroyed when the expression in which the call appears finishes. Destroying the temporary decrements the reference count, which goes to zero. The memory to which the temporary points is freed when the temporary is destroyed.

But `x` continues to point to that (freed) memory; `x` is now a dangling pointer. Attempting to use the value of `x` is undefined.

When we bind a `shared_ptr` to a plain pointer, we give responsibility for that memory to that `shared_ptr`. Once we give `shared_ptr` responsibility for a pointer, we should no longer use a built-in pointer to access the memory to which the `shared_ptr` now points.

WARNING It is dangerous to use a built-in pointer to access an object owned by a smart pointer, because we may not know when that object is destroyed.

. . . and Don’t Use `get` to Initialize or Assign Another Smart Pointer

The smart pointer types define a function named `get` (described in Table 12.1 (p. 452)) that returns a built-in pointer to the object that the smart pointer is managing. This function is intended for cases when we need to pass a built-in pointer to code that can’t use a smart pointer. The code that uses the return from `get` must not `delete` that pointer.

Although the compiler will not complain, it is an error to bind another smart pointer to the pointer returned by `get`:

```
shared_ptr<int> p(new int(42)); // reference count is 1
int *q = p.get();   // ok: but don't use q in any way that might delete its pointer
{ // new block
// undefined: two independent shared_ptrs point to the same memory
shared_ptr<int>(q);
} // block ends, q is destroyed, and the memory to which q points is freed
int foo = *p; // undefined; the memory to which p points was freed
```

In this case, both `p` and `q` point to the same memory. Because they were created independently from each other, each has a reference count of 1. When the block in which `q` was defined ends, `q` is destroyed. Destroying `q` frees the memory to which `q` points. That makes `p` into a dangling pointer, meaning that what happens when we attempt to use `p` is undefined. Moreover, when `p` is destroyed, the pointer to that memory will be `delete`d a second time.

WARNING Use `get` only to pass access to the pointer to code that you know will not `delete` the pointer. In particular, never use `get` to initialize or assign to another smart pointer.

Other `shared_ptr` Operations

The `shared_ptr` class gives us a few other operations, which are listed in Table 12.2 (p. 453) and Table 12.3 (on the previous page). We can use `reset` to assign a new pointer to a `shared_ptr`:

```
p = new int(1024);          // error: cannot assign a pointer to a shared_ptr
p.reset(new int(1024));     // ok: p points to a new object
```

Like assignment, `reset` updates the reference counts and, if appropriate, deletes the object to which `p` points. The `reset` member is often used together with `unique` to control changes to the object shared among several `shared_ptr`s. Before changing the underlying object, we check whether we’re the only user. If not, we make a new copy before making the change:

```
if (!p.unique())
    p.reset(new string(*p)); // we aren't alone; allocate a new copy
*p += newVal; // now that we know we're the only pointer, okay to change this object
```

Exercises Section 12.1.3

Exercise 12.10: Explain whether the following call to the `process` function defined on page 464 is correct. If not, how would you correct the call?

```
shared_ptr<int> p(new int(42));
process(shared_ptr<int>(p));
```

Exercise 12.11: What would happen if we called `process` as follows?

```
process(shared_ptr<int>(p.get()));
```

Exercise 12.12: Using the declarations of `p` and `sp` explain each of the following calls to `process`. If the call is legal, explain what it does. If the call is illegal, explain why:

```
auto p = new int();
auto sp = make_shared<int>();
(a) process(sp);
(b) process(new int());
(c) process(p);
(d) process(shared_ptr<int>(p));
```

Exercise 12.13: What happens if we execute the following code?

```
auto sp = make_shared<int>();
auto p = sp.get();
delete p;
```

12.1.4 Smart Pointers and Exceptions

In § 5.6.2 (p. 196) we noted that programs that use exception handling to continue processing after an exception occurs need to ensure that resources are properly freed if an exception occurs. One easy way to make sure resources are freed is to use smart pointers.

When we use a smart pointer, the smart pointer class ensures that memory is freed when it is no longer needed even if the block is exited prematurely:

```
void f()
{
    shared_ptr<int> sp(new int(42)); // allocate a new object
    // code that throws an exception that is not caught inside f
} // shared_ptr freed automatically when the function ends
```

When a function is exited, whether through normal processing or due to an exception, all the local objects are destroyed. In this case, `sp` is a `shared_ptr`, so destroying `sp` checks its reference count. Here, `sp` is the only pointer to the memory it manages; that memory will be freed as part of destroying `sp`.

In contrast, memory that we manage directly is not automatically freed when an exception occurs. If we use built-in pointers to manage memory and an exception occurs after a `new` but before the corresponding `delete`, then that memory won't be freed:

```
void f()
{
    int *ip = new int(42);       // dynamically allocate a new object
    // code that throws an exception that is not caught inside f
    delete ip;                   // free the memory before exiting
}
```

If an exception happens between the `new` and the `delete`, and is not caught inside `f`, then this memory can never be freed. There is no pointer to this memory outside the function `f`. Thus, there is no way to free this memory.

Smart Pointers and Dumb Classes

Many C++ classes, including all the library classes, define destructors (§ 12.1.1, p. 452) that take care of cleaning up the resources used by that object. However, not all classes are so well behaved. In particular, classes that are designed to be used by both C and C++ generally require the user to specifically free any resources that are used.

Classes that allocate resources—and that do not define destructors to free those resources—can be subject to the same kind of errors that arise when we use dynamic memory. It is easy to forget to release the resource. Similarly, if an exception happens between when the resource is allocated and when it is freed, the program will leak that resource.

We can often use the same kinds of techniques we use to manage dynamic memory to manage classes that do not have well-behaved destructors. For example, imagine we're using a network library that is used by both C and C++. Programs that use this library might contain code such as

```
struct destination;  // represents what we are connecting to
struct connection;   // information needed to use the connection
connection connect(destination*); // open the connection
void disconnect(connection);      // close the given connection
void f(destination &d /* other parameters */)
{
    // get a connection; must remember to close it when done
    connection c = connect(&d);
    // use the connection
    // if we forget to call disconnect before exiting f, there will be no way to close c
}
```

If `connection` had a destructor, that destructor would automatically close the connection when `f` completes. However, `connection` does not have a destructor. This problem is nearly identical to our previous program that used a `shared_ptr` to avoid memory leaks. It turns out that we can also use a `shared_ptr` to ensure that the `connection` is properly closed.

Using Our Own Deletion Code

By default, `shared_ptr`s assume that they point to dynamic memory. Hence, by default, when a `shared_ptr` is destroyed, it executes `delete` on the pointer it holds. To use a `shared_ptr` to manage a `connection`, we must first define a function to use in place of `delete`. It must be possible to call this **deleter** function with the pointer stored inside the `shared_ptr`. In this case, our deleter must take a single argument of type `connection*`:

```
void end_connection(connection *p) { disconnect(*p); }
```

When we create a `shared_ptr`, we can pass an optional argument that points to a deleter function (§ 6.7, p. 247):

```
void f(destination &d /* other parameters */)
{
    connection c = connect(&d);
    shared_ptr<connection> p(&c, end_connection);
    // use the connection
    // when f exits, even if by an exception, the connection will be properly closed
}
```

When `p` is destroyed, it won't execute `delete` on its stored pointer. Instead, `p` will call `end_connection` on that pointer. In turn, `end_connection` will call `disconnect`, thus ensuring that the connection is closed. If `f` exits normally, then `p` will be destroyed as part of the return. Moreover, `p` will also be destroyed, and the connection will be closed, if an exception occurs.

CAUTION: SMART POINTER PITFALLS

Smart pointers can provide safety and convenience for handling dynamically allocated memory only when they are used properly. To use smart pointers correctly, we must adhere to a set of conventions:

- **Don't use the same built-in pointer value to initialize (or `reset`) more than one smart pointer.**
- **Don't `delete` the pointer returned from `get()`.**
- **Don't use `get()` to initialize or `reset` another smart pointer.**
- **If you use a pointer returned by `get()`, remember that the pointer will become invalid when the last corresponding smart pointer goes away.**
- **If you use a smart pointer to manage a resource other than memory allocated by `new`, remember to pass a deleter (§ 12.1.4, p. 468, and § 12.1.5, p. 471).**

EXERCISES SECTION 12.1.4

Exercise 12.14: Write your own version of a function that uses a `shared_ptr` to manage a `connection`.

Exercise 12.15: Rewrite the first exercise to use a lambda (§ 10.3.2, p. 388) in place of the `end_connection` function.

12.1.5 `unique_ptr`

C++ 11

A `unique_ptr` "owns" the object to which it points. Unlike `shared_ptr`, only one `unique_ptr` at a time can point to a given object. The object to which a `unique_ptr` points is destroyed when the `unique_ptr` is destroyed. Table 12.4 lists the operations specific to `unique_ptr`s. The operations common to both were covered in Table 12.1 (p. 452).

Unlike `shared_ptr`, there is no library function comparable to `make_shared` that returns a `unique_ptr`. Instead, when we define a `unique_ptr`, we bind it to a pointer returned by `new`. As with `shared_ptr`s, we must use the direct form of initialization:

```
unique_ptr<double> p1; // unique_ptr that can point at a double
unique_ptr<int> p2(new int(42)); // p2 points to int with value 42
```

Because a `unique_ptr` owns the object to which it points, `unique_ptr` does not support ordinary copy or assignment:

```
unique_ptr<string> p1(new string("Stegosaurus"));
unique_ptr<string> p2(p1);  // error: no copy for unique_ptr
unique_ptr<string> p3;
p3 = p2;                    // error: no assign for unique_ptr
```

Table 12.4: `unique_ptr` Operations (See Also Table 12.1 (p. 452))

`unique_ptr<T> u1`
`unique_ptr<T, D> u2`
	Null `unique_ptr`s that can point to objects of type `T`. `u1` will use `delete` to free its pointer; `u2` will use a callable object of type `D` to free its pointer.

`unique_ptr<T, D> u(d)`	Null `unique_ptr` that point to objects of type `T` that uses `d`, which must be an object of type `D` in place of `delete`.
`u = nullptr`	Deletes the object to which `u` points; makes `u` null.
`u.release()`	Relinquishes control of the pointer `u` had held; returns the pointer `u` had held and makes `u` null.

`u.reset()`
`u.reset(q)`
`u.reset(nullptr)`
	Deletes the object to which `u` points;
	If the built-in pointer `q` is supplied, makes `u` point to that object.
	Otherwise makes `u` null.

Although we can't copy or assign a `unique_ptr`, we can transfer ownership from one (nonconst) `unique_ptr` to another by calling `release` or `reset`:

```
// transfers ownership from p1 (which points to the string Stegosaurus) to p2
unique_ptr<string> p2(p1.release()); // release makes p1 null
unique_ptr<string> p3(new string("Trex"));
// transfers ownership from p3 to p2
p2.reset(p3.release()); // reset deletes the memory to which p2 had pointed
```

The `release` member returns the pointer currently stored in the `unique_ptr` and makes that `unique_ptr` null. Thus, `p2` is initialized from the pointer value that had been stored in `p1` and `p1` becomes null.

The reset member takes an optional pointer and repositions the unique_ptr to point to the given pointer. If the unique_ptr is not null, then the object to which the unique_ptr had pointed is deleted. The call to reset on p2, therefore, frees the memory used by the string initialized from "Stegosaurus", transfers p3's pointer to p2, and makes p3 null.

Calling release breaks the connection between a unique_ptr and the object it had been managing. Often the pointer returned by release is used to initialize or assign another smart pointer. In that case, responsibility for managing the memory is simply transferred from one smart pointer to another. However, if we do not use another smart pointer to hold the pointer returned from release, our program takes over responsibility for freeing that resource:

```
p2.release(); // WRONG: p2 won't free the memory and we've lost the pointer
auto p = p2.release(); // ok, but we must remember to delete(p)
```

Passing and Returning unique_ptrs

There is one exception to the rule that we cannot copy a unique_ptr: We can copy or assign a unique_ptr that is about to be destroyed. The most common example is when we return a unique_ptr from a function:

```
unique_ptr<int> clone(int p) {
    // ok: explicitly create a unique_ptr<int> from int*
    return unique_ptr<int>(new int(p));
}
```

Alternatively, we can also return a copy of a local object:

```
unique_ptr<int> clone(int p) {
    unique_ptr<int> ret(new int (p));
    // ...
    return ret;
}
```

In both cases, the compiler knows that the object being returned is about to be destroyed. In such cases, the compiler does a special kind of "copy" which we'll discuss in § 13.6.2 (p. 534).

BACKWARD COMPATIBILITY: auto_ptr

Earlier versions of the library included a class named auto_ptr that had some, but not all, of the properties of unique_ptr. In particular, it was not possible to store an auto_ptr in a container, nor could we return one from a function.

Although auto_ptr is still part of the standard library, programs should use unique_ptr instead.

Passing a Deleter to unique_ptr

Like shared_ptr, by default, unique_ptr uses delete to free the object to which a unique_ptr points. As with shared_ptr, we can override the default deleter in a unique_ptr (§ 12.1.4, p. 468). However, for reasons we'll describe in § 16.1.6 (p. 676), the way unique_ptr manages its deleter is differs from the way shared_ptr does.

Overridding the deleter in a unique_ptr affects the unique_ptr type as well as how we construct (or reset) objects of that type. Similar to overriding the comparison operation of an associative container (§ 11.2.2, p. 425), we must supply the deleter type inside the angle brackets along with the type to which the unique_ptr can point. We supply a callable object of the specified type when we create or reset an object of this type:

```
// p points to an object of type objT and uses an object of type delT to free that object
// it will call an object named fcn of type delT
unique_ptr<objT, delT> p (new objT, fcn);
```

As a somewhat more concrete example, we'll rewrite our connection program to use a unique_ptr in place of a shared_ptr as follows:

```
void f(destination &d /* other needed parameters */)
{
    connection c = connect(&d);  // open the connection
    // when p is destroyed, the connection will be closed
    unique_ptr<connection, decltype(end_connection)*>
        p(&c, end_connection);
    // use the connection
    // when f exits, even if by an exception, the connection will be properly closed
}
```

Here we use decltype (§ 2.5.3, p. 70) to specify the function pointer type. Because decltype(end_connection) returns a function type, we must remember to add a * to indicate that we're using a pointer to that type (§ 6.7, p. 250).

EXERCISES SECTION 12.1.5

Exercise 12.16: Compilers don't always give easy-to-understand error messages if we attempt to copy or assign a unique_ptr. Write a program that contains these errors to see how your compiler diagnoses them.

Exercise 12.17: Which of the following unique_ptr declarations are illegal or likely to result in subsequent program error? Explain what the problem is with each one.

```
int ix = 1024, *pi = &ix, *pi2 = new int(2048);
typedef unique_ptr<int> IntP;
```

(a) `IntP p0(ix);` (b) `IntP p1(pi);`
(c) `IntP p2(pi2);` (d) `IntP p3(&ix);`
(e) `IntP p4(new int(2048));` (f) `IntP p5(p2.get());`

Exercise 12.18: Why doesn't shared_ptr have a release member?

12.1.6 `weak_ptr`

A `weak_ptr` (Table 12.5) is a smart pointer that does not control the lifetime of the object to which it points. Instead, a `weak_ptr` points to an object that is managed by a `shared_ptr`. Binding a `weak_ptr` to a `shared_ptr` does not change the reference count of that `shared_ptr`. Once the last `shared_ptr` pointing to the object goes away, the object itself will be deleted. That object will be deleted even if there are `weak_ptr`s pointing to it—hence the name `weak_ptr`, which captures the idea that a `weak_ptr` shares its object "weakly."

C++ 11

When we create a `weak_ptr`, we initialize it from a `shared_ptr`:

```
auto p = make_shared<int>(42);
weak_ptr<int> wp(p);   //  wp weakly shares with p; use count in p is unchanged
```

Here both `wp` and `p` point to the same object. Because the sharing is weak, creating `wp` doesn't change the reference count of `p`; it is possible that the object to which `wp` points might be deleted.

Because the object might no longer exist, we cannot use a `weak_ptr` to access its object directly. To access that object, we must call `lock`. The `lock` function checks whether the object to which the `weak_ptr` points still exists. If so, `lock` returns a `shared_ptr` to the shared object. As with any other `shared_ptr`, we are guaranteed that the underlying object to which that `shared_ptr` points continues to exist at least as long as that `shared_ptr` exists. For example:

```
if (shared_ptr<int> np = wp.lock()) { //  true if np is not null
    //  inside the if, np shares its object with p
}
```

Here we enter the body of the `if` only if the call to `lock` succeeds. Inside the `if`, it is safe to use `np` to access that object.

Table 12.5: `weak_ptr`s

`weak_ptr<T> w`	Null `weak_ptr` that can point at objects of type `T`.
`weak_ptr<T> w(sp)`	`weak_ptr` that points to the same object as the `shared_ptr` `sp`. `T` must be convertible to the type to which `sp` points.
`w = p`	`p` can be a `shared_ptr` or a `weak_ptr`. After the assignment `w` shares ownership with `p`.
`w.reset()`	Makes `w` null.
`w.use_count()`	The number of `shared_ptr`s that share ownership with `w`.
`w.expired()`	Returns `true` if `w.use_count()` is zero, `false` otherwise.
`w.lock()`	If `expired` is `true`, returns a null `shared_ptr`; otherwise returns a `shared_ptr` to the object to which `w` points.

Checked Pointer Class

As an illustration of when a `weak_ptr` is useful, we'll define a companion pointer class for our `StrBlob` class. Our pointer class, which we'll name `StrBlobPtr`, will store a `weak_ptr` to the `data` member of the `StrBlob` from which it was initialized. By using a `weak_ptr`, we don't affect the lifetime of the `vector` to which a given `StrBlob` points. However, we can prevent the user from attempting to access a `vector` that no longer exists.

`StrBlobPtr` will have two data members: `wptr`, which is either null or points to a `vector` in a `StrBlob`; and `curr`, which is the index of the element that this object currently denotes. Like its companion `StrBlob` class, our pointer class has a `check` member to verify that it is safe to dereference the `StrBlobPtr`:

```
// StrBlobPtr throws an exception on attempts to access a nonexistent element
class StrBlobPtr {
public:
    StrBlobPtr(): curr(0) { }
    StrBlobPtr(StrBlob &a, size_t sz = 0):
            wptr(a.data), curr(sz) { }
    std::string& deref() const;
    StrBlobPtr& incr();       // prefix version
private:
    // check returns a shared_ptr to the vector if the check succeeds
    std::shared_ptr<std::vector<std::string>>
        check(std::size_t, const std::string&) const;
    // store a weak_ptr, which means the underlying vector might be destroyed
    std::weak_ptr<std::vector<std::string>> wptr;
    std::size_t curr;      // current position within the array
};
```

The default constructor generates a null `StrBlobPtr`. Its constructor initializer list (§ 7.1.4, p. 265) explicitly initializes `curr` to zero and implicitly initializes `wptr` as a null `weak_ptr`. The second constructor takes a reference to `StrBlob` and an optional index value. This constructor initializes `wptr` to point to the `vector` in the `shared_ptr` of the given `StrBlob` object and initializes `curr` to the value of `sz`. We use a default argument (§ 6.5.1, p. 236) to initialize `curr` to denote the first element by default. As we'll see, the `sz` parameter will be used by the `end` member of `StrBlob`.

It is worth noting that we cannot bind a `StrBlobPtr` to a `const StrBlob` object. This restriction follows from the fact that the constructor takes a reference to a non`const` object of type `StrBlob`.

The `check` member of `StrBlobPtr` differs from the one in `StrBlob` because it must check whether the `vector` to which it points is still around:

```
std::shared_ptr<std::vector<std::string>>
StrBlobPtr::check(std::size_t i, const std::string &msg) const
{
    auto ret = wptr.lock();   // is the vector still around?
    if (!ret)
        throw std::runtime_error("unbound StrBlobPtr");
    if (i >= ret->size())
        throw std::out_of_range(msg);
    return ret; // otherwise, return a shared_ptr to the vector
}
```

Because a weak_ptr does not participate in the reference count of its corresponding shared_ptr, the vector to which this StrBlobPtr points might have been deleted. If the vector is gone, lock will return a null pointer. In this case, any reference to the vector will fail, so we throw an exception. Otherwise, check verifies its given index. If that value is okay, check returns the shared_ptr it obtained from lock.

Pointer Operations

We'll learn how to define our own operators in Chapter 14. For now, we've defined functions named deref and incr to dereference and increment the StrBlobPtr, respectively. The deref member calls check to verify that it is safe to use the vector and that curr is in range:

```
std::string& StrBlobPtr::deref() const
{
    auto p = check(curr, "dereference past end");
    return (*p)[curr];  // (*p) is the vector to which this object points
}
```

If check succeeds, p is a shared_ptr to the vector to which this StrBlobPtr points. The expression (*p)[curr] dereferences that shared_ptr to get the vector and uses the subscript operator to fetch and return the element at curr.

The incr member also calls check:

```
// prefix: return a reference to the incremented object
StrBlobPtr& StrBlobPtr::incr()
{
    // if curr already points past the end of the container, can't increment it
    check(curr, "increment past end of StrBlobPtr");
    ++curr;        // advance the current state
    return *this;
}
```

We'll also give our StrBlob class begin and end operations. These members will return StrBlobPtrs pointing to the first or one past the last element in the StrBlob itself. In addition, because StrBlobPtr accesses the data member of StrBlob, we must also make StrBlobPtr a friend of StrBlob (§ 7.3.4, p. 279):

```
class StrBlob {
    friend class StrBlobPtr;
    // other members as in § 12.1.1 (p. 456)
    StrBlobPtr begin(); // return StrBlobPtr to the first element
    StrBlobPtr end();   // and one past the last element
};
// these members can't be defined until StrStrBlob and StrStrBlobPtr are defined
StrBlobPtr StrBlob::begin() { return StrBlobPtr(*this); }
StrBlobPtr StrBlob::end()
    { return StrBlobPtr(*this, data->size()); }
```

Exercises Section 12.1.6

Exercise 12.19: Define your own version of `StrBlobPtr` and update your `StrBlob` class with the appropriate `friend` declaration and `begin` and `end` members.

Exercise 12.20: Write a program that reads an input file a line at a time into a `StrBlob` and uses a `StrBlobPtr` to print each element in that `StrBlob`.

Exercise 12.21: We could have written `StrBlobPtr`'s `deref` member as follows:

```
std::string& deref() const
{ return (*check(curr, "dereference past end"))[curr]; }
```

Which version do you think is better and why?

Exercise 12.22: What changes would need to be made to `StrBlobPtr` to create a class that can be used with a `const StrBlob`? Define a class named `ConstStrBlobPtr` that can point to a `const StrBlob`.

12.2 Dynamic Arrays

The `new` and `delete` operators allocate objects one at a time. Some applications, need the ability to allocate storage for many objects at once. For example, `vector`s and `string`s store their elements in contiguous memory and must allocate several elements at once whenever the container has to be reallocated (§ 9.4, p. 355).

To support such usage, the language and library provide two ways to allocate an array of objects at once. The language defines a second kind of `new` expression that allocates and initializes an array of objects. The library includes a template class named `allocator` that lets us separate allocation from initialization. For reasons we'll explain in § 12.2.2 (p. 481), using an `allocator` generally provides better performance and more flexible memory management.

Many, perhaps even most, applications have no direct need for dynamic arrays. When an application needs a varying number of objects, it is almost always easier, faster, and safer to do as we did with `StrBlob`: use a `vector` (or other library container). For reasons we'll explain in § 13.6 (p. 531), the advantages of using a library container are even more pronounced under the new standard. Libraries that support the new standard tend to be dramatically faster than previous releases.

Best Practices

Most applications should use a library container rather than dynamically allocated arrays. Using a container is easier, less likely to contain memory-management bugs, *and* is likely to give better performance.

As we've seen, classes that use the containers can use the default versions of the operations for copy, assignment, and destruction (§ 7.1.5, p. 267). Classes that allocate dynamic arrays must define their own versions of these operations to manage the associated memory when objects are copied, assigned, and destroyed.

WARNING

Do not allocate dynamic arrays in code inside classes until you have read Chapter 13.

12.2.1 `new` and Arrays

We ask `new` to allocate an array of objects by specifying the number of objects to allocate in a pair of square brackets after a type name. In this case, `new` allocates the requested number of objects and (assuming the allocation succeeds) returns a pointer to the first one:

```
// call get_size to determine how many ints to allocate
int *pia = new int[get_size()]; // pia points to the first of these ints
```

The size inside the brackets must have integral type but need not be a constant.

We can also allocate an array by using a type alias (§ 2.5.1, p. 67) to represent an array type. In this case, we omit the brackets:

```
typedef int arrT[42];   // arrT names the type array of 42 ints
int *p = new arrT;      // allocates an array of 42 ints; p points to the first one
```

Here, `new` allocates an array of `int`s and returns a pointer to the first one. Even though there are no brackets in our code, the compiler executes this expression using `new[]`. That is, the compiler executes this expression as if we had written

```
int *p = new int[42];
```

Allocating an Array Yields a Pointer to the Element Type

Although it is common to refer to memory allocated by `new T[]` as a "dynamic array," this usage is somewhat misleading. When we use `new` to allocate an array, we do not get an object with an array type. Instead, we get a pointer to the element type of the array. Even if we use a type alias to define an array type, `new` does not allocate an object of array type. In this case, the fact that we're allocating an array is not even visible; there is no `[`*num*`]`. Even so, `new` returns a pointer to the element type.

Because the allocated memory does not have an array type, we cannot call `begin` or `end` (§ 3.5.3, p. 118) on a dynamic array. These functions use the array dimension (which is part of an array's type) to return pointers to the first and one past the last elements, respectively. For the same reasons, we also cannot use a range `for` to process the elements in a (so-called) dynamic array. C++ 11

WARNING: It is important to remember that what we call a dynamic array does not have an array type.

Initializing an Array of Dynamically Allocated Objects

By default, objects allocated by `new`—whether allocated as a single object or in an array—are default initialized. We can value initialize (§ 3.3.1, p. 98) the elements in an array by following the size with an empty pair of parentheses.

```
int *pia = new int[10];             // block of ten uninitialized ints
int *pia2 = new int[10]();          // block of ten ints value initialized to 0

string *psa = new string[10];       // block of ten empty strings
string *psa2 = new string[10]();    // block of ten empty strings
```

C++ 11 Under the new standard, we can also provide a braced list of element initializers:

```
// block of ten ints each initialized from the corresponding initializer
int *pia3 = new int[10]{0,1,2,3,4,5,6,7,8,9};
// block of ten strings; the first four are initialized from the given initializers
// remaining elements are value initialized
string *psa3 = new string[10]{"a", "an", "the", string(3,'x')};
```

As when we list initialize an object of built-in array type (§ 3.5.1, p. 114), the initializers are used to initialize the first elements in the array. If there are fewer initializers than elements, the remaining elements are value initialized. If there are more initializers than the given size, then the `new` expression fails and no storage is allocated. In this case, `new` throws an exception of type `bad_array_new_length`. Like `bad_alloc`, this type is defined in the `new` header.

Although we can use empty parentheses to value initialize the elements of an array, we cannot supply an element initializer inside the parentheses. The fact that C++ 11 we cannot supply an initial value inside the parentheses means that we cannot use `auto` to allocate an array (§ 12.1.2, p. 459).

It Is Legal to Dynamically Allocate an Empty Array

We can use an arbitrary expression to determine the number of objects to allocate:

```
size_t n = get_size(); // get_size returns the number of elements needed
int* p = new int[n];    // allocate an array to hold the elements
for (int* q = p; q != p + n; ++q)
     /* process the array */ ;
```

An interesting question arises: What happens if `get_size` returns 0? The answer is that our code works fine. Calling `new[n]` with `n` equal to 0 is legal even though we cannot create an array variable of size 0:

```
char arr[0];              // error: cannot define a zero-length array
char *cp = new char[0]; // ok: but cp can't be dereferenced
```

When we use `new` to allocate an array of size zero, `new` returns a valid, nonzero pointer. That pointer is guaranteed to be distinct from any other pointer returned by `new`. This pointer acts as the off-the-end pointer (§ 3.5.3, p. 119) for a zero-element array. We can use this pointer in ways that we use an off-the-end iterator. The pointer can be compared as in the loop above. We can add zero to (or subtract zero from) such a pointer and can subtract the pointer from itself, yielding zero. The pointer cannot be dereferenced—after all, it points to no element.

In our hypothetical loop, if `get_size` returns 0, then `n` is also 0. The call to `new` will allocate zero objects. The condition in the `for` will fail (`p` is equal to `q + n` because `n` is 0). Thus, the loop body is not executed.

Freeing Dynamic Arrays

To free a dynamic array, we use a special form of `delete` that includes an empty pair of square brackets:

```
delete p;       // p must point to a dynamically allocated object or be null
delete [] pa;   // pa must point to a dynamically allocated array or be null
```

The second statement destroys the elements in the array to which pa points and frees the corresponding memory. Elements in an array are destroyed in reverse order. That is, the last element is destroyed first, then the second to last, and so on.

When we delete a pointer to an array, the empty bracket pair is essential: It indicates to the compiler that the pointer addresses the first element of an array of objects. If we omit the brackets when we delete a pointer to an array (or provide them when we delete a pointer to an object), the behavior is undefined.

Recall that when we use a type alias that defines an array type, we can allocate an array without using [] with new. Even so, we must use brackets when we delete a pointer to that array:

```
typedef int arrT[42];   // arrT names the type array of 42 ints
int *p = new arrT;      // allocates an array of 42 ints; p points to the first one
delete [] p;            // brackets are necessary because we allocated an array
```

Despite appearances, p points to the first element of an array of objects, not to a single object of type arrT. Thus, we must use [] when we delete p.

WARNING The compiler is unlikely to warn us if we forget the brackets when we delete a pointer to an array or if we use them when we delete a pointer to an object. Instead, our program is apt to misbehave without warning during execution.

Smart Pointers and Dynamic Arrays

The library provides a version of unique_ptr that can manage arrays allocated by new. To use a unique_ptr to manage a dynamic array, we must include a pair of empty brackets after the object type:

```
// up points to an array of ten uninitialized ints
unique_ptr<int[]> up(new int[10]);
up.release();    // automatically uses delete [] to destroy its pointer
```

The brackets in the type specifier (<int[]>) say that up points not to an int but to an array of ints. Because up points to an array, when up destroys the pointer it manages, it will automatically use delete[].

unqiue_ptrs that point to arrays provide slightly different operations than those we used in § 12.1.5 (p. 470). These operations are described in Table 12.6 (overleaf). When a unique_ptr points to an array, we cannot use the dot and arrow member access operators. After all, the unqiue_ptr points to an array, not an object so these operators would be meaningless. On the other hand, when a unqiue_ptr points to an array, we can use the subscript operator to access the elements in the array:

```
for (size_t i = 0; i != 10; ++i)
    up[i] = i;   // assign a new value to each of the elements
```

Table 12.6: `unique_ptr`s to Arrays

Member access operators (dot and arrow) are not supported for `unique_ptr`s to arrays. Other `unique_ptr` operations unchanged.

`unique_ptr<T[]> u`	`u` can point to a dynamically allocated array of type `T`.
`unique_ptr<T[]> u(p)`	`u` points to the dynamically allocated array to which the built-in pointer `p` points. `p` must be convertible to `T*` (§ 4.11.2, p. 161).
`u[i]`	Returns the object at position `i` in the array that `u` owns. **`u` must point to an array.**

Unlike `unique_ptr`, `shared_ptr`s provide no direct support for managing a dynamic array. If we want to use a `shared_ptr` to manage a dynamic array, we must provide our own deleter:

```
// to use a shared_ptr we must supply a deleter
shared_ptr<int> sp(new int[10], [](int *p) { delete[] p; });
sp.reset(); // uses the lambda we supplied that uses delete[] to free the array
```

Here we pass a lambda (§ 10.3.2, p. 388) that uses `delete[]` as the deleter.

Had we neglected to supply a deleter, this code would be undefined. By default, `shared_ptr` uses `delete` to destroy the object to which it points. If that object is a dynamic array, using `delete` has the same kinds of problems that arise if we forget to use `[]` when we delete a pointer to a dynamic array (§ 12.2.1, p. 479).

The fact that `shared_ptr` does not directly support managing arrays affects how we access the elements in the array:

```
// shared_ptrs don't have subscript operator and don't support pointer arithmetic
for (size_t i = 0; i != 10; ++i)
    *(sp.get() + i) = i; // use get to get a built-in pointer
```

There is no subscript operator for `shared_ptr`s, and the smart pointer types do not support pointer arithmetic. As a result, to access the elements in the array, we must use `get` to obtain a built-in pointer, which we can then use in normal ways.

EXERCISES SECTION 12.2.1

Exercise 12.23: Write a program to concatenate two string literals, putting the result in a dynamically allocated array of `char`. Write a program to concatenate two library `string`s that have the same value as the literals used in the first program.

Exercise 12.24: Write a program that reads a string from the standard input into a dynamically allocated character array. Describe how your program handles varying size inputs. Test your program by giving it a string of data that is longer than the array size you've allocated.

Exercise 12.25: Given the following `new` expression, how would you `delete` `pa`?

```
int *pa = new int[10];
```

12.2.2 The `allocator` Class

An aspect of `new` that limits its flexibility is that `new` combines allocating memory with constructing object(s) in that memory. Similarly, `delete` combines destruction with deallocation. Combining initialization with allocation is usually what we want when we allocate a single object. In that case, we almost certainly know the value the object should have.

When we allocate a block of memory, we often plan to construct objects in that memory as needed. In this case, we'd like to decouple memory allocation from object construction. Decoupling construction from allocation means that we can allocate memory in large chunks and pay the overhead of constructing the objects only when we actually need to create them.

In general, coupling allocation and construction can be wasteful. For example:

```
string *const p = new string[n]; // construct n empty strings
string s;
string *q = p;                   // q points to the first string
while (cin >> s && q != p + n)
    *q++ = s;                    // assign a new value to *q
const size_t size = q - p;       // remember how many strings we read
// use the array
delete[] p;   // p points to an array; must remember to use delete[]
```

This `new` expression allocates and initializes `n` `strings`. However, we might not need `n` `strings`; a smaller number might suffice. As a result, we may have created objects that are never used. Moreover, for those objects we do use, we immediately assign new values over the previously initialized `strings`. The elements that are used are written twice: first when the elements are default initialized, and subsequently when we assign to them.

More importantly, classes that do not have default constructors cannot be dynamically allocated as an array.

The `allocator` Class

The library **`allocator`** class, which is defined in the `memory` header, lets us separate allocation from construction. It provides type-aware allocation of raw, unconstructed, memory. Table 12.7 (overleaf) outlines the operations that `allocator` supports. In this section, we'll describe the `allocator` operations. In § 13.5 (p. 524), we'll see an example of how this class is typically used.

Like `vector`, `allocator` is a template (§ 3.3, p. 96). To define an `allocator` we must specify the type of objects that a particular `allocator` can allocate. When an `allocator` object allocates memory, it allocates memory that is appropriately sized and aligned to hold objects of the given type:

```
allocator<string> alloc;          // object that can allocate strings
auto const p = alloc.allocate(n); // allocate n unconstructed strings
```

This call to `allocate` allocates memory for `n` `strings`.

Table 12.7: Standard `allocator` Class and Customized Algorithms

`allocator<T> a`	Defines an `allocator` object named `a` that can allocate memory for objects of type `T`.
`a.allocate(n)`	Allocates raw, unconstructed memory to hold `n` objects of type `T`.
`a.deallocate(p, n)`	Deallocates memory that held `n` objects of type `T` starting at the address in the `T*` pointer `p`; `p` must be a pointer previously returned by `allocate`, and `n` must be the size requested when `p` was created. The user must run `destroy` on any objects that were constructed in this memory before calling `deallocate`.
`a.construct(p, `*args*`)`	`p` must be a pointer to type `T` that points to raw memory; *args* are passed to a constructor for type `T`, which is used to construct an object in the memory pointed to by `p`.
`a.destroy(p)`	Runs the destructor (§ 12.1.1, p. 452) on the object pointed to by the `T*` pointer `p`.

`allocators` Allocate Unconstructed Memory

The memory an `allocator` allocates is *unconstructed*. We use this memory by constructing objects in that memory. In the new library the `construct` member takes a pointer and zero or more additional arguments; it constructs an element at the given location. The additional arguments are used to initialize the object being constructed. Like the arguments to `make_shared` (§ 12.1.1, p. 451), these additional arguments must be valid initializers for an object of the type being con-
C++ 11 structed. In particular, if the , object is a class type, these arguments must match a constructor for that class:

```
auto q = p; //  q will point to one past the last constructed element
alloc.construct(q++);            //  *q is the empty string
alloc.construct(q++, 10, 'c');   //  *q is cccccccccc
alloc.construct(q++, "hi");      //  *q is hi!
```

In earlier versions of the library, `construct` took only two arguments: the pointer at which to construct an object and a value of the element type. As a result, we could only copy an element into unconstructed space, we could not use any other constructor for the element type.

It is an error to use raw memory in which an object has not been constructed:

```
cout << *p << endl;   //  ok: uses the string output operator
cout << *q << endl;   //  disaster: q points to unconstructed memory!
```

WARNING We must `construct` objects in order to use memory returned by `allocate`. Using unconstructed memory in other ways is undefined.

When we're finished using the objects, we must destroy the elements we constructed, which we do by calling `destroy` on each constructed element. The `destroy` function takes a pointer and runs the destructor (§ 12.1.1, p. 452) on the pointed-to object:

```
    while (q != p)
        alloc.destroy(--q);          // free the strings we actually allocated
```

At the beginning of our loop, q points one past the last constructed element. We decrement q before calling destroy. Thus, on the first call to destroy, q points to the last constructed element. We destroy the first element in the last iteration, after which q will equal p and the loop ends.

WARNING

We may destroy only elements that are actually constructed.

Once the elements have been destroyed, we can either reuse the memory to hold other strings or return the memory to the system. We free the memory by calling deallocate:

```
    alloc.deallocate(p, n);
```

The pointer we pass to deallocate cannot be null; it must point to memory allocated by allocate. Moreover, the size argument passed to deallocate must be the same size as used in the call to allocate that obtained the memory to which the pointer points.

Algorithms to Copy and Fill Uninitialized Memory

As a companion to the allocator class, the library also defines two algorithms that can construct objects in uninitialized memory. These functions, described in Table 12.8, are defined in the memory header.

Table 12.8: `allocator` Algorithms

These functions construct elements in the destination, rather than assigning to them.

`uninitialized_copy(b, e, b2)`	Copies elements from the input range denoted by iterators b and e into unconstructed, raw memory denoted by the iterator b2. The memory denoted by b2 must be large enough to hold a copy of the elements in the input range.
`uninitialized_copy_n(b, n, b2)`	Copies n elements starting from the one denoted by the iterator b into raw memory starting at b2.
`uninitialized_fill(b, e, t)`	Constructs objects in the range of raw memory denoted by iterators b and e as a copy of t.
`uninitialized_fill_n(b, n, t)`	Constructs an unsigned number n objects starting at b. b must denote unconstructed, raw memory large enough to hold the given number of objects.

As an example, assume we have a vector of ints that we want to copy into dynamic memory. We'll allocate memory for twice as many ints as are in the vector. We'll construct the first half of the newly allocated memory by copying elements from the original vector. We'll construct elements in the second half by filling them with a given value:

```
// allocate twice as many elements as vi holds
auto p = alloc.allocate(vi.size() * 2);
// construct elements starting at p as copies of elements in vi
auto q = uninitialized_copy(vi.begin(), vi.end(), p);
// initialize the remaining elements to 42
uninitialized_fill_n(q, vi.size(), 42);
```

Like the `copy` algorithm (§ 10.2.2, p. 382), `uninitialized_copy` takes three iterators. The first two denote an input sequence and the third denotes the destination into which those elements will be copied. The destination iterator passed to `uninitialized_copy` must denote unconstructed memory. Unlike `copy`, `uninitialized_copy` constructs elements in its destination.

Like `copy`, `uninitialized_copy` returns its (incremented) destination iterator. Thus, a call to `uninitialized_copy` returns a pointer positioned one element past the last constructed element. In this example, we store that pointer in `q`, which we pass to `uninitialized_fill_n`. This function, like `fill_n` (§ 10.2.2, p. 380), takes a pointer to a destination, a count, and a value. It will construct the given number of objects from the given value at locations starting at the given destination.

Exercises Section 12.2.2

Exercise 12.26: Rewrite the program on page 481 using an `allocator`.

12.3 Using the Library: A Text-Query Program

To conclude our discussion of the library, we'll implement a simple text-query program. Our program will let a user search a given file for words that might occur in it. The result of a query will be the number of times the word occurs and a list of lines on which that word appears. If a word occurs more than once on the same line, we'll display that line only once. Lines will be displayed in ascending order—that is, line 7 should be displayed before line 9, and so on.

For example, we might read the file that contains the input for this chapter and look for the word `element`. The first few lines of the output would be

```
element occurs 112 times
   (line 36) A set element contains only a key;
   (line 158) operator creates a new element
   (line 160) Regardless of whether the element
   (line 168) When we fetch an element from a map, we
   (line 214) If the element is not found, find returns
```

followed by the remaining 100 or so lines in which the word `element` occurs.

12.3.1 Design of the Query Program

A good way to start the design of a program is to list the program's operations. Knowing what operations we need can help us see what data structures we'll need. Starting from requirements, the tasks our program must do include the following:

- When it reads the input, the program must remember the line(s) in which each word appears. Hence, the program will need to read the input a line at a time and break up the lines from the input file into its separate words
- When it generates output,
  - The program must be able to fetch the line numbers associated with a given word
  - The line numbers must appear in ascending order with no duplicates
  - The program must be able to print the text appearing in the input file at a given line number.

These requirements can be met quite neatly by using various library facilities:

- We'll use a `vector<string>` to store a copy of the entire input file. Each line in the input file will be an element in this `vector`. When we want to print a line, we can fetch the line using its line number as the index.
- We'll use an `istringstream` (§ 8.3, p. 321) to break each line into words.
- We'll use a `set` to hold the line numbers on which each word in the input appears. Using a `set` guarantees that each line will appear only once and that the line numbers will be stored in ascending order.
- We'll use a `map` to associate each word with the `set` of line numbers on which the word appears. Using a `map` will let us fetch the `set` for any given word.

For reasons we'll explain shortly, our solution will also use `shared_ptr`s.

Data Structures

Although we could write our program using `vector`, `set`, and `map` directly, it will be more useful if we define a more abstract solution. We'll start by designing a class to hold the input file in a way that makes querying the file easy. This class, which we'll name `TextQuery`, will hold a `vector` and a `map`. The `vector` will hold the text of the input file; the `map` will associate each word in that file to the `set` of line numbers on which that word appears. This class will have a constructor that reads a given input file and an operation to perform the queries.

The work of the query operation is pretty simple: It will look inside its `map` to see whether the given word is present. The hard part in designing this function is deciding what the query function should return. Once we know that a word was found, we need to know how often it occurred, the line numbers on which it occurred, and the corresponding text for each of those line numbers.

The easiest way to return all those data is to define a second class, which we'll name `QueryResult`, to hold the results of a query. This class will have a `print` function to print the results in a `QueryResult`.

Sharing Data between Classes

Our QueryResult class is intended to represent the results of a query. Those results include the set of line numbers associated with the given word and the corresponding lines of text from the input file. These data are stored in objects of type TextQuery.

Because the data that a QueryResult needs are stored in a TextQuery object, we have to decide how to access them. We could copy the set of line numbers, but that might be an expensive operation. Moreover, we certainly wouldn't want to copy the vector, because that would entail copying the entire file in order to print (what will usually be) a small subset of the file.

We could avoid making copies by returning iterators (or pointers) into the TextQuery object. However, this approach opens up a pitfall: What happens if the TextQuery object is destroyed before a corresponding QueryResult? In that case, the QueryResult would refer to data in an object that no longer exists.

This last observation about synchronizing the lifetime of a QueryResult with the TextQuery object whose results it represents suggests a solution to our design problem. Given that these two classes conceptually "share" data, we'll use shared_ptrs (§ 12.1.1, p. 450) to reflect that sharing in our data structures.

Using the TextQuery Class

When we design a class, it can be helpful to write programs using the class before actually implementing the members. That way, we can see whether the class has the operations we need. For example, the following program uses our proposed TextQuery and QueryResult classes. This function takes an ifstream that points to the file we want to process, and interacts with a user, printing the results for the given words:

```
void runQueries(ifstream &infile)
{
    // infile is an ifstream that is the file we want to query
    TextQuery tq(infile);  // store the file and build the query map
    // iterate with the user: prompt for a word to find and print results
    while (true) {
        cout << "enter word to look for, or q to quit: ";
        string s;
        // stop if we hit end-of-file on the input or if a 'q' is entered
        if (!(cin >> s) || s == "q") break;
        // run the query and print the results
        print(cout, tq.query(s)) << endl;
    }
}
```

We start by initializing a TextQuery object named tq from a given ifstream. The TextQuery constructor reads that file into its vector and builds the map that associates the words in the input with the line numbers on which they appear.

The while loop iterates (indefinitely) with the user asking for a word to query and printing the related results. The loop condition tests the literal true (§ 2.1.3, p. 41), so it always succeeds. We exit the loop through the break (§ 5.5.1, p. 190) after the first `if`. That `if` checks that the read succeeded. If so, it also checks whether the user entered a `q` to quit. Once we have a word to look for, we ask `tq` to find that word and then call `print` to print the results of the search.

EXERCISES SECTION 12.3.1

Exercise 12.27: The `TextQuery` and `QueryResult` classes use only capabilities that we have already covered. Without looking ahead, write your own versions of these classes.

Exercise 12.28: Write a program to implement text queries without defining classes to manage the data. Your program should take a file and interact with a user to query for words in that file. Use `vector`, `map`, and `set` containers to hold the data for the file and to generate the results for the queries.

Exercise 12.29: We could have written the loop to manage the interaction with the user as a `do while` (§ 5.4.4, p. 189) loop. Rewrite the loop to use a `do while`. Explain which version you prefer and why.

12.3.2 Defining the Query Program Classes

We'll start by defining our `TextQuery` class. The user will create objects of this class by supplying an `istream` from which to read the input file. This class also provides the `query` operation that will take a `string` and return a `QueryResult` representing the lines on which that `string` appears.

The data members of the class have to take into account the intended sharing with `QueryResult` objects. The `QueryResult` class will share the `vector` representing the input file and the `set`s that hold the line numbers associated with each word in the input. Hence, our class has two data members: a `shared_ptr` to a dynamically allocated `vector` that holds the input file, and a `map` from `string` to `shared_ptr<set>`. The `map` associates each word in the file with a dynamically allocated `set` that holds the line numbers on which that word appears.

To make our code a bit easier to read, we'll also define a type member (§ 7.3.1, p. 271) to refer to line numbers, which are indices into a `vector` of `string`s:

```
class QueryResult; // declaration needed for return type in the query function
class TextQuery {
public:
    using line_no = std::vector<std::string>::size_type;
    TextQuery(std::ifstream&);
    QueryResult query(const std::string&) const;
private:
    std::shared_ptr<std::vector<std::string>> file; // input file
    // map of each word to the set of the lines in which that word appears
    std::map<std::string,
             std::shared_ptr<std::set<line_no>>> wm;
};
```

The hardest part about this class is untangling the class names. As usual, for code that will go in a header file, we use `std::` when we use a library name (§ 3.1, p. 83). In this case, the repeated use of `std::` makes the code a bit hard to read at first. For example,

```
std::map<std::string, std::shared_ptr<std::set<line_no>>> wm;
```

is easier to understand when rewritten as

```
map<string, shared_ptr<set<line_no>>> wm;
```

The `TextQuery` Constructor

The `TextQuery` constructor takes an `ifstream`, which it reads a line at a time:

```
// read the input file and build the map of lines to line numbers
TextQuery::TextQuery(ifstream &is): file(new vector<string>)
{
    string text;
    while (getline(is, text)) {        // for each line in the file
        file->push_back(text);         // remember this line of text
        int n = file->size() - 1;      // the current line number
        istringstream line(text);      // separate the line into words
        string word;
        while (line >> word) {         // for each word in that line
            // if word isn't already in wm, subscripting adds a new entry
            auto &lines = wm[word]; // lines is a shared_ptr
            if (!lines) // that pointer is null the first time we see word
                lines.reset(new set<line_no>); // allocate a new set
            lines->insert(n);       // insert this line number
        }
    }
}
```

The constructor initializer allocates a new `vector` to hold the text from the input file. We use `getline` to read the file a line at a time and push each line onto the `vector`. Because `file` is a `shared_ptr`, we use the `->` operator to dereference `file` to fetch the `push_back` member of the `vector` to which `file` points.

Next we use an `istringstream` (§ 8.3, p. 321) to process each word in the line we just read. The inner `while` uses the `istringstream` input operator to read each word from the current line into `word`. Inside the `while`, we use the `map` subscript operator to fetch the `shared_ptr<set>` associated with `word` and bind `lines` to that pointer. Note that `lines` is a reference, so changes made to `lines` will be made to the element in `wm`.

If `word` wasn't in the `map`, the subscript operator adds `word` to `wm` (§ 11.3.4, p. 435). The element associated with `word` is value initialized, which means that `lines` will be a null pointer if the subscript operator added `word` to `wm`. If `lines` is null, we allocate a `new set` and call `reset` to update the `shared_ptr` to which `lines` refers to point to this newly allocated `set`.

Regardless of whether we created a new `set`, we call `insert` to add the current line number. Because `lines` is a reference, the call to `insert` adds an element to the set in wm. If a given word occurs more than once in the same line, the call to insert does nothing.

The QueryResult Class

The QueryResult class has three data members: a string that is the word whose results it represents; a shared_ptr to the vector containing the input file; and a shared_ptr to the set of line numbers on which this word appears. Its only member function is a constructor that initializes these three members:

```
class QueryResult {
friend std::ostream& print(std::ostream&, const QueryResult&);
public:
    QueryResult(std::string s,
                std::shared_ptr<std::set<line_no>> p,
                std::shared_ptr<std::vector<std::string>> f):
        sought(s), lines(p), file(f) { }
private:
    std::string sought;  // word this query represents
    std::shared_ptr<std::set<line_no>> lines; // lines it's on
    std::shared_ptr<std::vector<std::string>> file;  // input file
};
```

The constructor's only job is to store its arguments in the corresponding data members, which it does in the constructor initializer list (§ 7.1.4, p. 265).

The query Function

The query function takes a string, which it uses to locate the corresponding set of line numbers in the map. If the string is found, the query function constructs a QueryResult from the given string, the TextQuery file member, and the set that was fetched from wm.

The only question is: What should we return if the given string is not found? In this case, there is no set to return. We'll solve this problem by defining a local static object that is a shared_ptr to an empty set of line numbers. When the word is not found, we'll return a copy of this shared_ptr:

```
QueryResult
TextQuery::query(const string &sought) const
{
    // we'll return a pointer to this set if we don't find sought
    static shared_ptr<set<line_no>> nodata(new set<line_no>);
    // use find and not a subscript to avoid adding words to wm!
    auto loc = wm.find(sought);
    if (loc == wm.end())
        return QueryResult(sought, nodata, file);  // not found
    else
        return QueryResult(sought, loc->second, file);
}
```

Printing the Results

The `print` function prints its given `QueryResult` object on its given stream:

```
ostream &print(ostream & os, const QueryResult &qr)
{
    // if the word was found, print the count and all occurrences
    os << qr.sought << " occurs " << qr.lines->size() << " "
       << make_plural(qr.lines->size(), "time", "s") << endl;
    // print each line in which the word appeared
    for (auto num : *qr.lines) // for every element in the set
        // don't confound the user with text lines starting at 0
        os << "\t(line " << num + 1 << ") "
           << *(qr.file->begin() + num) << endl;
    return os;
}
```

We use the `size` of the `set` to which the `qr.lines` points to report how many matches were found. Because that `set` is in a `shared_ptr`, we have to remember to dereference `lines`. We call `make_plural` (§ 6.3.2, p. 224) to print `time` or `times`, depending on whether that size is equal to 1.

In the `for` we iterate through the `set` to which `lines` points. The body of the `for` prints the line number, adjusted to use human-friendly counting. The numbers in the `set` are indices of elements in the `vector`, which are numbered from zero. However, most users think of the first line as line number 1, so we systematically add 1 to the line numbers to convert to this more common notation.

We use the line number to fetch a line from the `vector` to which `file` points. Recall that when we add a number to an iterator, we get the element that many elements further into the `vector` (§ 3.4.2, p. 111). Thus, `file->begin() + num` is the `num`th element after the start of the `vector` to which `file` points.

Note that this function correctly handles the case that the word is not found. In this case, the `set` will be empty. The first output statement will note that the word occurred 0 times. Because `*res.lines` is empty. the `for` loop won't be executed.

EXERCISES SECTION 12.3.2

Exercise 12.30: Define your own versions of the `TextQuery` and `QueryResult` classes and execute the `runQueries` function from § 12.3.1 (p. 486).

Exercise 12.31: What difference(s) would it make if we used a `vector` instead of a `set` to hold the line numbers? Which approach is better? Why?

Exercise 12.32: Rewrite the `TextQuery` and `QueryResult` classes to use a `StrBlob` instead of a `vector<string>` to hold the input file.

Exercise 12.33: In Chapter 15 we'll extend our query system and will need some additional members in the `QueryResult` class. Add members named `begin` and `end` that return iterators into the `set` of line numbers returned by a given query, and a member named `get_file` that returns a `shared_ptr` to the file in the `QueryResult` object.

Chapter Summary

In C++, memory is allocated through `new` expressions and freed through `delete` expressions. The library also defines an `allocator` class for allocating blocks of dynamic memory.

Programs that allocate dynamic memory are responsible for freeing the memory they allocate. Properly freeing dynamic memory is a rich source of bugs: Either the memory is never freed, or it is freed while there are still pointers referring to the memory. The new library defines smart pointers—`shared_ptr`, `unique_ptr`, and `weak_ptr`—that make managing dynamic memory much safer. A smart pointer automatically frees the memory once there are no other users of that memory. When possible, modern C++ programs ought to use smart pointers.

Defined Terms

allocator Library class that allocates unconstructed memory.

dangling pointer A pointer that refers to memory that once had an object but no longer does. Program errors due to dangling pointers are notoriously difficult to debug.

delete Frees memory allocated by `new`. `delete p` frees the object and `delete [] p` frees the array to which `p` points. `p` may be null or point to memory allocated by `new`.

deleter Function passed to a smart pointer to use in place of `delete` when destroying the object to which the pointer is bound.

destructor Special member function that cleans up an object when the object goes out of scope or is deleted.

dynamically allocated Object that is allocated on the free store. Objects allocated on the free store exist until they are explicitly deleted or the program terminates.

free store Memory pool available to a program to hold dynamically allocated objects.

heap Synonym for free store.

new Allocates memory from the free store. `new T` allocates and constructs an object of type `T` and returns a pointer to that object; if `T` is an array type, `new` returns a pointer to the first element in the array. Similarly, `new [`*n*`] T` allocates *n* objects of type `T` and returns a pointer to the first element in the array. By default, the allocated object is default initialized. We may also provide optional initializers.

placement new Form of `new` that takes additional arguments passed in parentheses following the keyword `new`; for example, `new (nothrow) int` tells `new` that it should not throw an exception.

reference count Counter that tracks how many users share a common object. Used by smart pointers to know when it is safe to delete memory to which the pointers point.

shared_ptr Smart pointer that provides shared ownership: The object is deleted when the last `shared_ptr` pointing to that object is destroyed.

smart pointer Library type that acts like a pointer but can be checked to see whether it is safe to use. The type takes care of deleting memory when appropriate.

unique_ptr Smart pointer that provides single ownership: The object is deleted when the `unique_ptr` pointing to that object is destroyed. `unique_ptrs` cannot be directly copied or assigned.

weak_ptr Smart pointer that points to an object managed by a `shared_ptr`. The `shared_ptr` does not count `weak_ptrs` when deciding whether to delete its object.