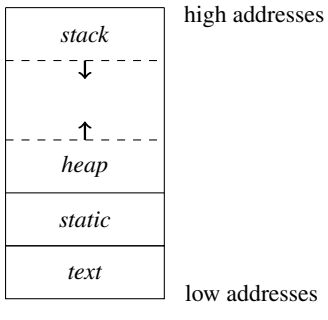
Chapter 27
Smart Pointers and Memory Management

27.1 Memory Allocation and Storage Duration Classes (⁎)

※ 27.1.1 *The Memory Model* (⁎)

When a process first begins running on your machine, it is assigned memory by the machine's *operating system*, which manages the physical hardware resources that a running process is able to use. This procedure of assigning memory to running processes is fairly complex, especially since hundreds of processes may be running at the same time on a single machine, so we will not discuss it in detail here. Instead, we will focus on the memory management procedures of a single running process and the mechanisms it can use to allocate memory for its own program data.

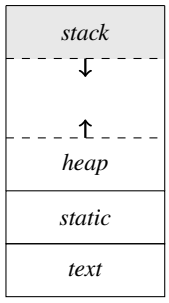
As we touched upon in the first chapter, every running process on a machine has its own memory layout. This memory layout depicts the address space of that process, which is used to store information the process may need to run. An example of a typical layout is shown below:



Each of these regions correspond to a different method of storage, of which include automatic, static, and dynamic storage. We will go over each of these storage categories in greater detail in this section.

※ 27.1.2 *The Stack and Automatic Storage* (✻)

The stack region of the address space is used for **automatic storage**. Objects that are created in automatic storage are locally scoped — that is, their lifetime is restricted within the scope they are initialized in. For more on scope, see section 1.7.

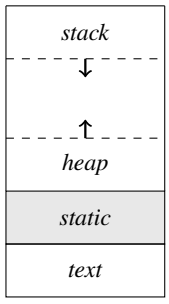


Attempting to reference a local variable outside of its scope would result in undefined behavior. The lifetime of objects allocated in automatic storage is maintained by the compiler, so you as the programmer will not need to be responsible for cleaning up the memory allocated for these objects. Some example code is shown below; here, `x`, `y`, and `z` are all initialized on the stack with the following local scopes:

```
1  void func(int32_t x) {
2    int32_t y = 281;
3    {
4      int32_t z = x + y;
5    }  // end scope of z
6  } // end scope of x and y
```

※ 27.1.3 *Global Variables and Static Storage* (✻)

Objects allocated in **static storage** may be expected to last throughout the duration of an entire program, and thus should be accessible at any point during a program's lifetime. Global variables, static member variables, and static local variables are all types of variables that may be allocated in static storage. These variables only have a single copy in memory that is referenced by the entire program.



Some example code involving global and static variables is shown below:

```
1  int32_t GLOBAL_VAR = 281;
2
3  struct Foo {
4    static const int32_t static_member = 370;
5  };
6
7  int32_t increment() {
8    static int32_t counter = GLOBAL_VAR;
9    return ++counter;
10 } // increment()
11
12 int main() {
13   std::cout << GLOBAL_VAR << '\n';          // prints 281
14   std::cout << Foo::static_member << '\n';  // prints 370
15   std::cout << increment() << '\n';         // prints 282
16   std::cout << increment() << '\n';         // prints 283
17 } // main()
```

Here, `GLOBAL_VAR` is a global variable, `Foo::static_member` is a static member variable, and `counter` is a static local variable. All three of these objects live in the static region of the address space, and they are accessible for the entire duration of the program.

Similar to objects in automatic storage, objects in static storage are also managed by the compiler. When the compiler translates your C++ source code into a machine-code executable that can be run on a machine, it is able to identify the variables that should be placed in static storage duration. Storage for these variables is then allocated at the start of the program (although the actual initialization of these variables may be deferred to when they are first used, depending on the programming language).

※ 27.1.4 *Linking Global Variables Across Multiple Translation Units* (✻)

If a global variable needs to be referenced for an entire program, what happens if you have multiple implementation files? In C++, each symbol (e.g., variable or function name) can be *declared* multiple times within its scope, but it can only be *defined* once (this is known as the *one definition rule (ODR)*). Declaring a symbol is akin to introducing the symbol with bare bones information needed to establish its existence, while defining a symbol actually specifies instructions on how it can be created. A comparison between the two is provided with the examples below:

Declaration:	**Definition:**
`int32_t x;`	`int32_t x{281};`
`void func(int32_t y);`	`void func(int32_t y) { std::cout << y << '\n'; }`
`struct Z;`	`struct Z { int32_t z1; double z2; };`

It may be possible for a program to consist of multiple *translation units*, which is a unit that comprises an implementation file (e.g., `.cpp`) along with all the headers (e.g., `.h`, `.hpp`) that it includes directly or indirectly. The compiler compiles each translation unit independently on its own, and then links the results together into a single executable that can be run. Most global variables exhibit a property known as *external linkage* by default, in that they can be visible from any translation unit in a program — thus, no two global objects within the same program can share the same name. If you want a global variable to exhibit *internal linkage* (i.e., the variable is only visible within its translation unit or implementation file), you can specify it as `static` to restrict its use to the unit it is declared in (as briefly covered in section 1.8.3).

That being said, even though a global variable may be visible to all the `.cpp` implementation files of a program, it can only be defined once due to the one definition rule. To satisfy this rule, you can use the `extern` keyword to indicate that the definition of a symbol exists in another translation unit. For non-const global variables, this is done by declaring *and* defining the variable normally in one file, and then declaring the variable in all the other files in the program with the `extern` keyword (and no definition). An example is shown below:

file1.cpp

```
// file1.cpp

int32_t GLOBAL_VAR = 281;  // declare and define (defining can only be done once)

/*
  ... other code ...
*/
```

file2.cpp

```
// file2.cpp (another file to be compiled for the same program)

// cannot redefine GLOBAL_VAR due to ODR, so specify it as extern to let the
// compiler know that the definition can be found in another translation unit
extern int32_t GLOBAL_VAR;

/*
  ... other code ...
*/

void func() {
  std::cout << GLOBAL_VAR << '\n';  // prints 281
} // func()
```

There is one notable exception: objects declared as `const` or `constexpr` exhibit internal linkage by default, so constant global variables can only be visible within the translation unit it is defined in. To allow a `const` global variable to exhibit external linkage, you will need to apply the `extern` keyword to both the definition as well as all declarations in any other files used by the program. An example is shown below:

file1.cpp

```
// file1.cpp

extern const int32_t GLOBAL_VAR = 281;  // declare and define (extern needed since this is const)

/*
  ... other code ...
*/
```

file2.cpp

```
// file2.cpp (another file to be compiled for the same program)

extern const int32_t GLOBAL_VAR;

/*
  ... other code ...
*/

void func() {
  std::cout << GLOBAL_VAR << '\n';  // prints 281
} // func()
```

On the topic of `extern`, if you ever work with C++ in industry, you may also end up seeing functions that are qualified as `extern "C"`, such as the one shown below:

```
extern "C" int foo(/* ... args ... */) {
  /*
    ... code here ...
  */
} // foo()
```

This indicates to the compiler that the function should be compiled and linked using C naming conventions. This is necessary because C and C++ compilation is not exactly the same. When compiling and linking C code, each function can be uniquely identified by its name. However, C++ cannot do this, as it supports function overloading, where functions with different parameters can share the *same* name. To account for this, C++ compilers use a technique known as *name mangling* to generate a unique identifier symbol for each function, even if multiple functions may share the same name. By using `extern "C"` to mark a function, you are essentially instructing the compiler to use that function's name as its own identifier symbol without any additional mangling (although this implies that these functions cannot be overloaded).

This is not really something you will need to worry about in most cases. That being said, there are scenarios where this keyword can be useful. One such example occurs if you are working on a shared library whose functions can be dynamically loaded at runtime. By defining a function with `extern "C"`, you prevent the compiler from name mangling the function name during compilation, allowing you to dynamically link and use that function from another executable, regardless of how that executable was compiled. Note that `extern "C"` does not imply that the code must be written in C — in fact, it can be used to dynamically link C++ libraries that may have been compiled with separate C++ compilers that mangle names in different ways (as the process used for name mangling is not defined by the standard).

Remark: C++20 introduced the concept of *modules*, which make it easier to share functionality across different translation units. With a module, you can compile shared functionality independently from the translation units that need them. By compiling each module as a separate unit, you are then able to import them more easily from other components, reduce compilation times, and avoid many existing problems associated with header files. Each module only needs to be compiled once, and the compiler is able to reuse its binary file wherever that module is imported into a project. A simple example using modules is shown below:

`helloworld.ixx`

```
// helloworld.ixx (module file)
#include <iostream>

export module helloworld;  // module declaration

export void print_hello_world() {
  std::cout << "Hello World!\n";
} // print_hello_world()
```

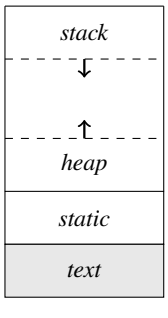
`main.cpp`

```
// main.cpp
import helloworld;          // import declaration

int main() {
  print_hello_world();      // prints "Hello World!"
} // main()
```

There is a lot to explore about modules, and we will not be able to cover it all here. You are definitely encouraged to learn more about modules on your own if you are curious about how they work.

※ 27.1.5 *Executable Instructions and Text Storage* (✻)

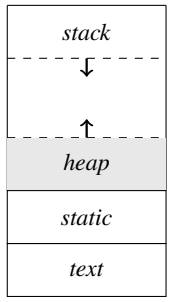
The text segment is a fixed-size segment of memory that is used to store executable instructions provided by the compiler.



This section typically contains the compiled machine code, the functions that make up the program, and other read-only instructional data. When a program begins execution, the contents of this section are read to determine how the program should run.

※ 27.1.6 *The Heap and Dynamic Storage* (✻)

For most of this chapter, we will be primarily concerned with a region of memory known as the heap. This portion of memory is used for **dynamic storage**, where objects can be explicitly created and destroyed by the programmer. An object is allocated in dynamic storage using the C++ operators of `new` and `new[]` (for arrays), and it is deallocated using the C++ operators of `delete` and `delete[]` (for arrays).



Some example code that uses dynamic memory is shown below:

```
1   int main() {
2     int32_t* ptr = new int32_t{281};     // allocates memory for an integer (281) on the heap
3     std::cout << *ptr << '\n';           // prints 281
4     delete ptr;                          // deletes the memory allocated for the integer 281
5
6     int32_t* arr = new int32_t[281];     // allocates memory for an array of size 281
7     for (size_t i = 0; i < 281; ++i) {
8       arr[i] = 281;                      // populates the array with the value 281
9     } // for i
10    delete[] arr;                        // deletes the memory for the array of size 281
11  } // main()
```

By allocating objects in dynamic memory, you ensure that they can still remain valid outside the scope in which they were created (and will remain so until they are explicitly deleted). In the example below, the object created by the function can still be used after the function exits. This can be useful if you want a function to create objects of multiple derived types based on runtime information (this is known as a *factory pattern*).

```
1   int32_t* get_number(int32_t number) {
2     int32_t* ptr = new int32_t{number};
3     return ptr;
4   } // get_object()
5
6   int main() {
7     int32_t* ptr = get_number(281);
8     std::cout << *ptr << '\n';           // prints 281
9     delete ptr;
10  } // main()
```

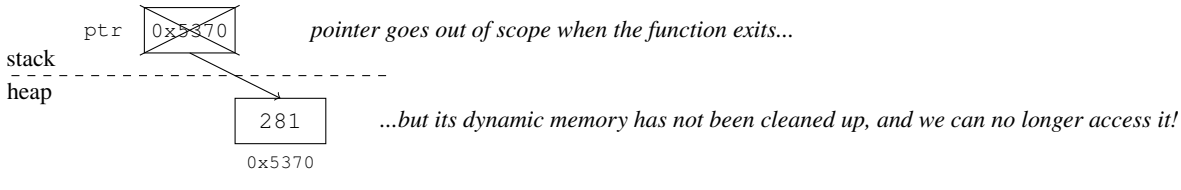
In C++, objects allocated in dynamic memory are a frequent source of bugs when they are not handled correctly. Unlike objects with static or automatic storage durations, whose lifetimes are managed by the compiler, objects allocated with dynamic memory must be explicitly managed by the programmer. If the programmer fails to properly manage this memory, then issues may arise. Examples of common issues involving dynamic memory include memory leaks, double deletions, and dangling pointers.

※ 27.1.7 *Common Issues Involving Dynamic Memory: Memory Leaks* (✻)

A **memory leak** is an error that occurs when a program fails to deallocate memory that it no longer needs. This typically happens when a pointer to dynamic memory goes out of scope before the memory it pointed to is cleaned up. This memory is thereby orphaned, and there is no longer any way to access it. The orphaned object sits in memory and cannot be cleaned up until the leaking program terminates, at which its memory is reclaimed by the operating system. A simple example of a memory leak is shown in the example code below:

```
1   void leaky_func() {
2     int32_t* ptr = new int32_t{281};
3     std::cout << *ptr << '\n';
4   } // leaky_func()
```

In the above code, dynamic memory is allocated on line 2 to store the integer `281`. However, `delete` is never called on this memory, so it is never cleaned up. When `ptr` goes out of scope on line 4, access to this dynamic memory is lost and therefore leaked.



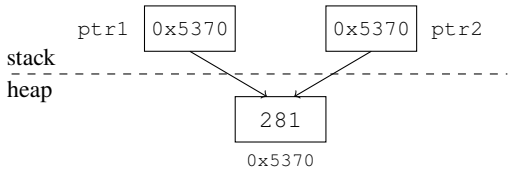
Memory leaks are one of the most common dynamic error bugs, and they can be a nuisance to deal with when they occur. A program exhibiting a memory leak may run fine for a while, leading to a false sense of security that there are no problems. However, as more and more memory gets leaked as time goes by, the amount of available memory decreases, and the program's performance deteriorates until it eventually crashes. Luckily, there are many tools available (such as Valgrind) that allow you to detect memory leaks in your code, and the usage of smart pointers can help you write safer code that is less prone to memory leaks. We will go over smart pointers in a later section of this chapter.

※ 27.1.8 *Common Issues Involving Dynamic Memory: Double Deletions* (✻)

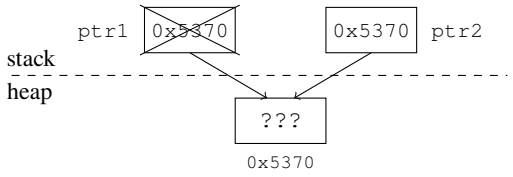
A **double delete**, also known as a double free, is a dynamic memory error that occurs when a program attempts to delete an object in dynamic memory more than once. An example of a double delete is shown below:

```
1  void double_delete() {
2    int32_t* ptr1 = new int32_t{281};
3    int32_t* ptr2 = ptr1;
4    delete ptr1;
5    delete ptr2;
6  } // double_delete()
```

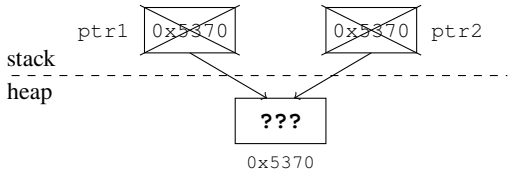
In this code, both `ptr1` and `ptr2` point to the same object in dynamic memory. When `ptr1` is deleted on line 4, the object it points to is cleaned up and invalidated. However, `ptr2` now points to the deallocated object, so deleting `ptr2` on line 5 tries to delete the same object again. This results in a double delete, which produces undefined behavior.



delete ptr1



***delete ptr2** (ERROR! Memory is already deleted!)*



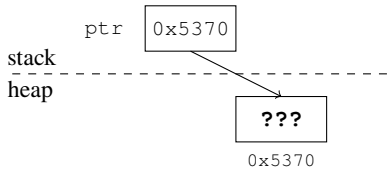
Double deletions can be dangerous if you introduce them in your code. While you may get a runtime crash when they happen, you may also corrupt other memory that may have been allocated at the location of the previously deallocated object. The outcome of a double delete is unpredictable because, as mentioned, its behavior is undefined. To avoid double deletions, you must be vigilant when transferring ownership of objects allocated in dynamic memory, so that two entities will not ever delete the same object. This is also something that can be prevented with the usage of smart pointers, as we will see later.

※ 27.1.9 *Common Issues Involving Dynamic Memory: Dangling Pointers* (✻)

A **dangling pointer** is a pointer that points to a dead object after it has already been deleted. If you try to dereference a dangling pointer, you would get undefined behavior. An example of a dangling pointer is shown below:

```
1  void dangling_pointer() {
2    int32_t* ptr = new int32_t{281};
3    std::cout << *ptr << '\n';        // prints 281
4    delete ptr;
5    std::cout << *ptr << '\n';        // undefined behavior, ptr is a dangling pointer!
6  } // dangling_pointer()
```

In this case, the object that `ptr` points to is deleted on line 4. However, line 5 still tries to reference the value of the object, which has already been invalidated. There is no saying what could be at the memory address that was deleted: perhaps the memory could have been reused for something else. As a result, the output of line 5 is undefined.



If you know that a pointer to dynamic memory will last longer than the object it points to in dynamic memory, a good strategy is to set it to `nullptr` after you call `delete` on it. By doing this, you remove the possibility of dereferencing an invalid address in memory. Note that dereferencing a `nullptr` would still be undefined behavior, but this can be detected more easily when it happens (and potentially give you a benign segmentation fault instead of potentially corrupting other memory). Setting a deleted pointer to `nullptr` can also reduce the risk of a double delete, since deleting a `nullptr` does nothing by default (and is thus safe to do).

※ 27.1.10 *Garbage Collection* (✻)

Languages such as C and C++ require the programmer to manage their own dynamic memory. As a result, if you explicitly create an object in dynamic memory, you must also remember to delete the object and free its memory after you are done using it. Although giving the programmer the responsibility of managing their own memory has its benefits (as it provides more flexibility and control), it can also lead to memory bugs like the ones we discussed previously and can make the language harder to learn. To address this, many other languages (such as Python, Java, and C#) utilize a technique known as **garbage collection** to automatically detect and free up memory that is no longer being used. When using these garbage-collected languages, you do not need to worry about managing dynamic memory manually — a garbage collection process is performed in the background to reclaim memory that you no longer need. Many strategies for garbage collection exist, but we will not be discussing all of them here (although one strategy, reference counting, will be covered when we introduce shared pointers in a later section).

27.2 Resource Acquisition is Initialization (RAII) (✻)

※ 27.2.1 *Motivations for RAII* (✻)

In C++, the programmer is responsible for managing their own dynamic memory. This should be simple, right? Whenever you allocate memory on the heap, just remember to also clean up this memory after you are done using it. In a perfect world, this would not be an issue… but alas, the world is not perfect, and a solution of saying *"just remember it"* is not an effective way to prevent potentially dangerous memory issues. In fact, dynamic memory problems may not even be caused by a programmer's carelessness or negligence: consider the following function, which upon first glance appears to abide by the rules of proper memory management:

```
1  void foo() {
2    int32_t* ptr = new int32_t{281};
3    bar(ptr);
4    /* ... other stuff ... */
5    delete ptr;
6  } // foo()
```

However, what happens if `bar()` throws an exception that propagates to `foo()`? Even though the programmer remembered to `delete ptr` on line 5, this statement is never reached due to the exception thrown on line 3. Thus, the function terminates without cleaning up the memory allocated for `ptr`, resulting in a memory leak. As a result, simply relying on a programmer to appropriately call `delete` is not enough to prevent all memory issues from occuring. Instead, as savvy developers, a better method would be to design our code in a way to prevent these issues from happening at all, regardless of how careful a programmer is while managing dynamically allocated memory.

To begin exploring how this can be done, let us look back at the lifetime of a local object that is allocated on the stack. When the local object is created within a scope, its constructor is automatically called, and when it goes out of scope, its destructor is automatically called. What if we could somehow give ownership of dynamically-allocated memory to an object that is automatically allocated and deallocated on the stack? All we have to do is allocate the dynamic memory in the object's constructor (using `new`) and deallocate the dynamic memory in the object's destructor (using `delete`). Then, when we create that object as a local variable on the stack, its destructor would be automatically called when it goes out of scope, which in turn frees the dynamic memory in its destructor. That way, dynamically allocated resources will always be deallocated automatically when they go out of scope, and there would be no need to explicitly delete this memory on your own!

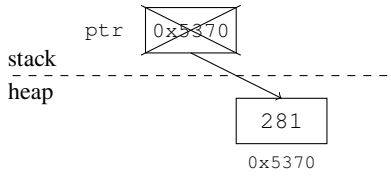
This idiom is known as **resource acquisition is initialization**, or **RAII**. With RAII, an object that needs to use a resource (such as dynamic memory) acquires it when it is initialized (via its constructor) and releases it when it is destroyed (via its destructor). The idea here is that all resources need to be released at some point, and a lot of common issues are caused by resources being maintained even when they are no longer used. We can ensure that resources are managed properly throughout our program by tying their usage directly with the lifetime of any object that needs them. RAII is not just a concept that is relevant to dynamic memory: it also applies to other resources that need to be shared, such as network connections and locks (which is a mechanism that can be used to restrict access to a shared resource that may be needed by multiple running threads at the same time — you will learn more about this if you take a class that covers concurrency).

※ 27.2.2 *Managing Dynamic Memory Using RAII* (✻)

Let us look at an example of RAII in action. Consider our initial example of a leaky function, where we allocate an object on the heap and then forget to delete it.

```
1  void foo() {
2    int32_t* ptr = new int32_t{281};
3    std::cout << *ptr << '\n';       // prints 281
4  } // foo()
```

As previously indicated, dynamic memory is allocated on line 2 to store the integer, but `delete` is never called on this memory. When `ptr` goes out of scope on line 4, this memory is leaked.



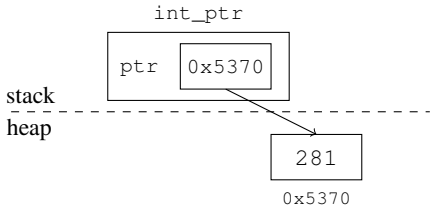
However, we can fix this problem by defining a separate class that can be used to manage this pointer. This class takes in the pointer upon construction and automatically deletes it in its destructor. We will also overload additional operators to allow our class to behave as if it were a pointer (supporting dereferencing using the $\star$ and $->$ operators). This is shown below:

```
1  class IntPtr {
2  public:
3    // constructor, takes in the value to allocate dynamic memory for
4    IntPtr(int32_t val) : ptr{new int32_t{val}} {}
5
6    // destructor, automatically deletes the pointer when it is invoked
7    ~IntPtr() {
8      delete ptr;
9    } // ~IntPtr()
10
11   // other members to make this object behave as if it were the pointer itself
12   int32_t& operator*() {
13     return *ptr;
14   } // operator*()
15
16   int32_t* operator->() {
17     return ptr;
18   } // operator->()
19
20 private:
21   // stores the pointer itself
22   int32_t* ptr;
23 };
```

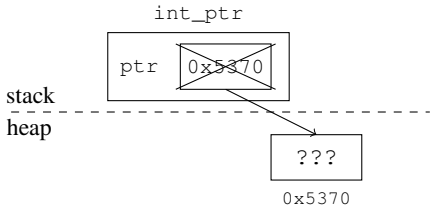
Instead of allocating the pointer directly, we can encapsulate it within the `IntPtr` class. The updated code is shown below:

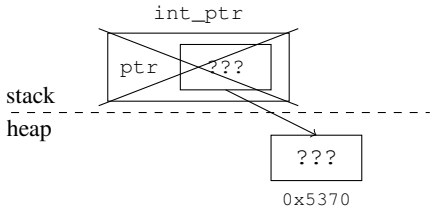
```
1  void foo() {
2    IntPtr int_ptr{281};
3    std::cout << *int_ptr << '\n'; // prints 281
4  } // foo()
```

Unlike the previous implementation, this function no longer leaks any memory. This is because the dynamic memory used by this function is managed by the `IntPtr` class, which is initialized as a local variable on the stack. When the `IntPtr` is initialized on line 2, it allocates dynamic memory within its constructor. Then, when it goes out of scope on line 4, it frees that dynamic memory within its destructor. Since the compiler ensures that constructors and destructors are called automatically for local variables when they are created and go out of scope, the programmer will not have to directly manage the dynamic memory that is needed by the function themselves!



`int_ptr` goes out of scope, which causes its destructor to be invoked — the destructor frees the dynamic memory stored in `int_ptr`





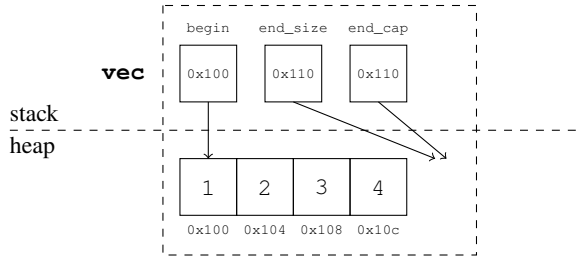
RAII can be particularly helpful if exceptions are involved. Recall our previous example, where seemingly responsible code that called `delete` after `new` still ended up leaking memory due to the presence of an exception.

```
void foo() {
  int32_t* ptr = new int32_t{281};
  bar(ptr);    // throws exception
  /* ... other code ... */
  delete ptr;  // this line is never reached!
} // foo()
```

If we used an `IntPtr` instead, this would no longer be a problem. If the `bar()` method throws an exception that propagates up to `foo()`, the `foo()` function would terminate, and `int_ptr` would be automatically deallocated from automatic storage (since it is a local variable). This in turn calls the destructor of `int_ptr`, which successfully cleans up the dynamic memory that was allocated.

```
void foo() {
  IntPtr int_ptr{281};
  bar(ptr);
  /* ... other code ... */
} // foo()
```

RAII as a concept is not actually new: we actually employed this principle while designing our dynamic array back in chapter 6. In fact, the idea of RAII is utilized by many containers in the STL. Consider the `std::vector<>` as an example. When you instantiate a vector of values, the values themselves are stored in dynamic memory by default (only bookkeeping data is located on the stack — see chapter 7 for more information on how vectors work).



However, when you instantiate a `std::vector<>`, you do not need to worry about any of this dynamic memory. This is because all dynamic memory is handled internally by the vector. If the vector needs any more memory, it allocates a new array using `new[]`. Then, when the vector goes out of scope, its destructor cleans up its dynamically allocated array using `delete[]`. This is the beauty of RAII: by delegating management of a resource to a local object via its constructor and destructor, we remove the need for the programmer to directly manage this resource on their own. This in turn reduces the frequency of bugs and makes it easier to write safer and cleaner code.

As we demonstrated with our initial `IntPtr` example, RAII can be used to encapsulate a pointer to dynamic memory within a local object, allowing the pointer to free its memory automatically when it goes out of scope via its destructor. This premise forms the foundation of *smart pointers*, which are class objects that can be used to manage dynamic memory in a similar manner. We will discuss smart pointers in greater detail in the following sections of this chapter.

27.3 Introduction to Smart Pointers (⁎)

※ 27.3.1 *Anatomy of a Smart Pointer* (⁎)

A **smart pointer** in C++ is a class object that can be used to manage objects in dynamic memory, automatically deleting the dynamically allocated memory for you at the appropriate time. When a smart pointer goes out of scope or has its contents reassigned, any dynamic memory that is no longer needed gets automatically cleaned up (using RAII principles). This prevents many of the issues discussed earlier, such as unintended memory leaks, double deletions, and dangling pointers.

Smart pointers are designed to mimic a normal pointer, with support for standard pointer operators such as `operator*` and `operator->`. Because of this, smart pointers can be substituted for raw pointers whenever you are working with dynamic memory, since they be used just like one (with the added bonus of automatic dynamic memory cleanup). You can think of a smart pointer as a templated class that stores an internal pointer to dynamic memory, with a destructor that handles the deletion of this dynamic memory (note that this is an oversimplification):

```
template <typename T>
class SmartPtr {
private:
  // internal pointer that points to dynamic memory
  T* ptr;
public:
  // destructor deletes this dynamic memory
  ~SmartPtr() {
    // DISCLAIMER: This does not work if this dynamic object is shared by multiple smart pointers!
    // We will address this issue later, but we can only delete if nothing else is using this memory.
    delete ptr;
  } // ~SmartPtr()
};
```

Smart pointers also have overloaded operators that allow them to behave like a normal pointer, allowing you to use them in place of a normal raw pointer when working with dynamic memory:

```
template <typename T>
class SmartPtr {
private:
  // internal pointer that points to dynamic memory
  T* ptr;
public:
  // overloaded dereference operator
  T& operator*() { return *ptr; }

  // overloaded indirection operator
  T* operator->() { return ptr; }
};
```

Smart pointers also provide a method for retrieving the internal pointer that it manages. When using smart pointers in C++, you can return the stored pointer by using the supplied `get()` method (although you have to be careful when using this pointer, since you could undermine the behavior of the smart pointer if you do anything bad with it).

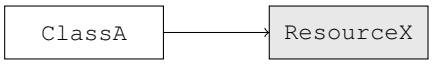
```
template <typename T>
class SmartPtr {
private:
  // internal pointer that points to dynamic memory
  T* ptr;
public:
  // returns the internal pointer to dynamic memory
  T* get() { return ptr; }
};
```

Lastly, it should be emphasized that smart pointers are used to manage *dynamically* allocated memory. You should not be using smart pointers to manage local objects on the stack, since those objects are already managed automatically by the compiler. Furthermore, because smart pointers attempt to `delete` their internal pointer after they are destructed, storing a pointer to a stack object would result in an error (since `delete` can only be called on objects allocated with `new`). Luckily, there are methods that can be used to check for this condition at compile time.

※ 27.3.2 *Categories of Ownership* (✻)

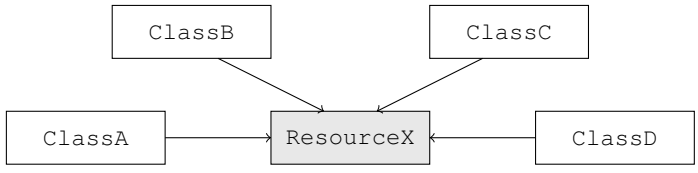
When working with smart pointers, one important factor to consider is the *ownership* of the dynamic memory that you want to manage. In C++, ownership of dynamic memory is given to the entity that is responsible for calling `delete` on the pointer after it is done using it. Prior to the advent of smart pointers, ownership could at times be difficult to discern, since a raw pointer on its own does not provide any information about whether it should be cleaned up by the entity that is using it (since you may not know if the object it points to is still being used by something else). Smart pointers were designed to fix this problem, as they can act as a clear source of ownership for an object in dynamic memory — a smart pointer accepts all responsibility for (1) keeping track of the number of entities that are using the object at any point in time and (2) deleting the memory for the object after it is no longer needed.

There are two major categories of ownership that we will discuss. The first (and most common) category is *exclusive ownership*. As its name implies, a dynamically allocated object under exclusive ownership only has one owner that is responsible for deleting it. An example of exclusive ownership is shown in the figure below. Here, a single class instance, `ClassA`, is using the dynamically allocated resource `ResourceX`. `ResourceX` is not needed by any other class instances, so when `ClassA` gets destructed, then `ResourceX` can be safely deleted as well.



Exclusive ownership is the simplest form of ownership to implement for a smart pointer. If you can ensure that only one entity owns a smart pointer at any point in time, then you can implement the smart pointer like we did previously in this section, since it would be safe to delete the pointer's dynamic memory after its owner no longer needs it.

A second, more complicated form of ownership is *shared ownership*. With shared ownership, an object in dynamic memory may be jointly owned by multiple entities at the same time. In this case, the object can only be deleted if all of its owners no longer need it. An example is demonstrated below, where `ResourceX` is referenced by multiple class instances.

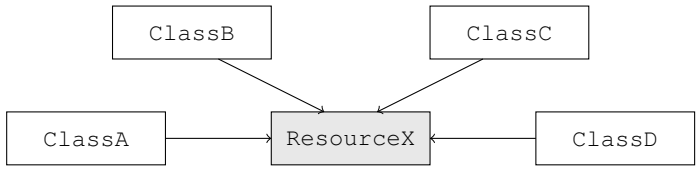


In this example, `ClassA`, `ClassB`, `ClassC`, and `ClassD` all need to use the shared `ResourceX`. If one of these class instances goes out of scope, the resource should *not* be deleted, since there would still be three additional class instances that require the resource (in fact, deleting an object under shared ownership using our previous implementation would result in a double delete, since the same resource would be deleted multiple times). Only when all four class instances go out of scope can the resource be safely deallocated.

For a smart pointer to support multiple owners, it must apply additional logic to keep track of how many objects still need to use the resource it is managing, so that it can correctly identify when to delete the resource. This is done using a technique known as *reference counting*.

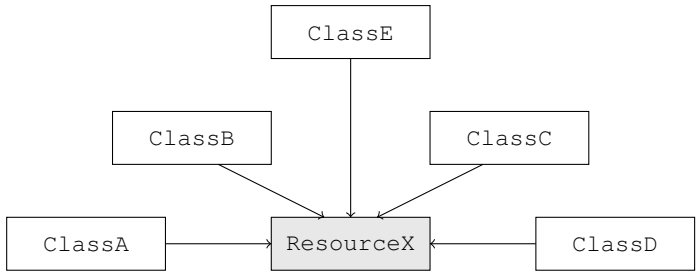
※ 27.3.3 *Reference Counting* (✻)

Reference counting is a strategy that can be used to manage resources that may be under shared ownership. With reference counting, a smart pointer keeps track of the number of references to a shared object in dynamic memory. If a new entity needs to use the shared object, the reference count of the object is incremented. Similarly, if an existing entity no longer needs to use the shared object, the reference count of the object is decremented. Once the shared object reaches a reference count of zero, its memory is cleaned up by the smart pointer. To illustrate this process, consider our previous example. In this case, four different entities need to use `ResourceX`, so it has a reference count of four.



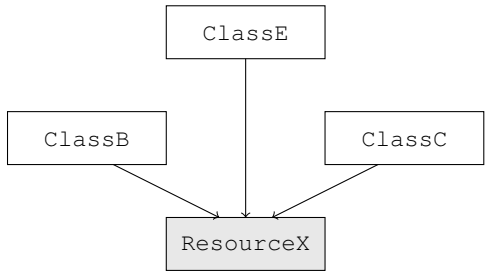
reference count: 4

Suppose we create a new instance of a class, `ClassE`, that initializes a smart pointer that also references `ResourceX`. When this happens, the reference count for `ResourceX` is incremented to five.



reference count: 5

Now, suppose the instances of `ClassA` and `ClassD` go out of scope and no longer need to use `ResourceX`. When this happens, the reference count of `ResourceX` drops to three.

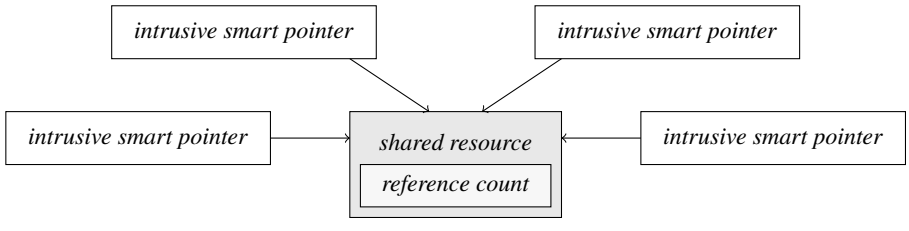


reference count: 3

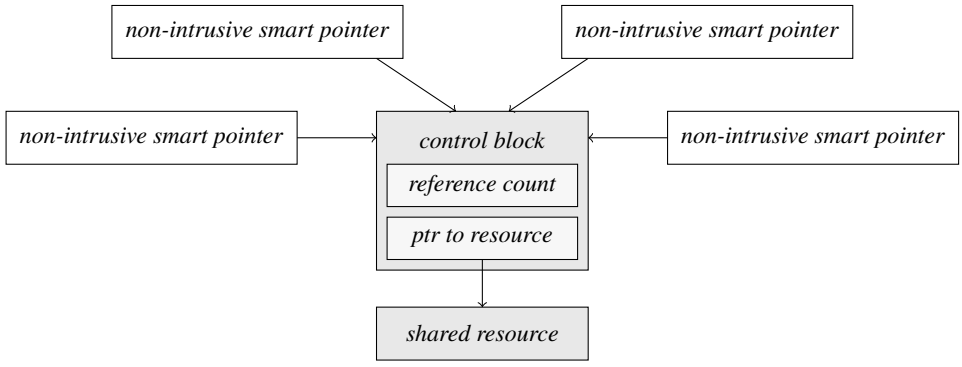
Eventually, the remaining class instances will go out of scope. Once this happens, there would no longer be any instances that still need to use `ResourceX`, so its reference count would drop to zero. The final smart pointer would detect this and subsequently clean up its memory.

reference count: 0

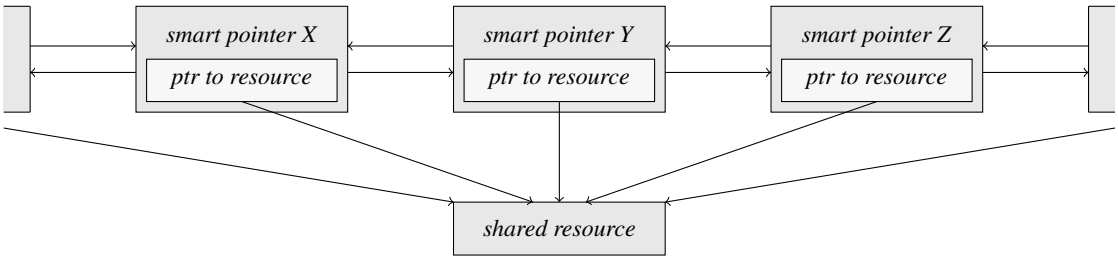
How can we implement reference counting when designing a smart pointer that can handle shared objects? There are two primary strategies that can be used: intrusive reference counting and non-intrusive reference counting. With **intrusive reference counting**, the pointed to object must keep track of the reference count itself. When a new smart pointer references a shared object (either through creation or assignment), the object increments its reference count. When an existing smart pointer no longer needs to reference a shared object, the object decrements its reference count. Then, when the reference count becomes zero, the object is deleted. This is typically done by implementing a new class that derives from the type of the shared object and adding a reference count member variable to it. An illustration of intrusive reference counting is shown below.



On the other hand, with **non-intrusive reference counting**, the shared resource does not need to contain a reference counter itself. Instead, the reference count is identified through another means without altering the shared resource. The most common implementation of non-intrusive reference counting is to allocate a separate *control block* that manages the reference counter of a shared resource in dynamic memory. In this implementation, the first smart pointer that points to a shared resource dynamically allocates a new control block that contains (1) a pointer to the shared object and (2) its reference count. This control block then has its reference count incremented and decremented as the number of smart pointer references changes. Once the reference count reaches zero, both the control block and the shared resource are automatically deleted. An illustration of this implementation is shown below.



Another non-intrusive reference counting implementation uses a doubly-linked list to identify when a shared object can be deleted. In this implementation, smart pointers that reference the same shared object are placed together in a linked list. When a new smart pointer needs to access the shared object, it gets spliced into the list. When an existing smart point no longer needs to access the shared object, it gets spliced out of the list. When the last smart pointer in the list is removed, it deletes the shared resource. Note that this implementation does not actually keep track of the actual reference count, but this is okay, since we only need to know whether the reference count is zero or non-zero.



Of these three methods, the control block implementation is the one that is typically the most practical. This is because it is best able to handle a specific edge case that occurs when a cycle of objects have smart pointers that reference each other.[1] This edge case is known as the *cycle problem*, and we will discuss it in more detail when we cover weak pointers in a later section.

27.4 Unique Pointers (⁎)

※ 27.4.1 *std::unique_ptr* (⁎)

The C++ `<memory>` library provides several implementations of smart pointers that you can use to manage dynamic memory. The simplest of these smart pointers is the `std::unique_ptr<>`, which can be used to manage a dynamically allocated object under *exclusive* ownership. An instance of a `std::unique_ptr<>` is very lightweight, as it basically just stores the pointer to dynamic memory that it manages (or `nullptr` if it does not own any dynamic memory). When a `std::unique_ptr<>` is destructed, its destructor deletes the dynamic memory referenced by its internal pointer — this is safe because a unique pointer can only be owned by a single entity at any point in time. In addition, like all the other smart pointers in the `<memory>` library, a unique pointer can be used syntatically as if it were a normal, built-in pointer (e.g., dereferencing, implicit conversion to a Boolean when placed in a conditional check such as an `if` statement, etc.).

There are two ways to initialize a `std::unique_ptr<>`. One method to initialize a unique pointer is to pass a pointer to dynamic memory directly into its constructor. This is shown in the example code below:

```
// initialize a unique pointer to a dynamically allocated integer
std::unique_ptr<int32_t> ptr{new int32_t{281}};
```

However, there are a few disadvantages with this approach, the most notable of which involves exception safety. Consider the following function, which takes in a unique pointer and an integer:

```
void foo(std::unique_ptr<Object> object, int32_t next_id);
```

[1] The control block version is also easier to implement when multithreading is involved; i.e., when multiple parts of a program are run at the same time and may each try to modify smart pointers that reference the same shared object. While the intrusive pointer and linked list versions can also be adapted to address the cycle problem and efficient multithreading, it is tougher to do so with these approaches.

Assume that there is another function, `get_next_id()`, that can be used to get the next available ID to be used in the program. Knowing this, the following function call should be valid:

```
foo(std::unique_ptr<Object>(new Object{}), get_next_id());
```

From this function call, it is reasonable to assume that the following steps must happen before `foo()` is invoked:

1. A new `Object` is dynamically allocated on the heap via `new`.
2. This new `Object` is passed into the constructor of the unique pointer.
3. The `get_next_id()` function is run to get the next available ID.

However, when converting this function call to machine code, the compiler is not required to construct the function arguments in any specific order. It is entirely acceptable for the steps to be executed in this order:

1. A new `Object` is dynamically allocated on the heap via `new`.
2. The `get_next_id()` function is run to get the next available ID.
3. This new `Object` is passed into the constructor of the unique pointer.

Now, consider what happens if the `get_next_id()` function throws an exception in step 2. Since the `Object` has already been dynamically allocated in step 1 but has not yet been passed into the constructor, it ends up being leaked! This is certainly not we want.

C++14 fixed this issue with the introduction of `std::make_unique()`, which is a better way to initialize a unique pointer.[2] The `std::make_unique()` method takes in the constructor arguments of the object to be dynamically allocated and forwards it directly to the object's constructor. The function signature of `std::make_unique()` is shown below.

```
template <typename T, typename... Args>
std::unique_ptr<T> std::make_unique(Args&&... args);
```

Constructs an object of type `T` using its constructor arguments `args` and wraps it in a `std::unique_ptr<>`.

Using `std::make_unique()`, we can create the same unique pointer to an integer as follows:

```
// initialize a unique pointer to a dynamic allocated integer
std::unique_ptr<int32_t> ptr = std::make_unique<int32_t>(281);
```

Additionally, when `std::make_unique()` is called, the dynamically allocated object is guaranteed to be stored in the smart pointer without any interruptions. This matters for the ID example, which previously leaked memory if the `get_next_id()` threw an exception after the new object was dynamically allocated, but before it could be stored in the smart pointer. `std::make_unique()` prevents this from happening.

```
// better, does not leak memory if an exception is thrown in get_next_id()
foo(std::make_unique<Object>(), get_next_id());
```

To create a unique pointer that manages a custom object, you should pass in the arguments of the object's constructor into `std::make_unique()`. An example is shown below.

```
1  struct Object {
2    int32_t member1;
3    bool member2;
4    Object(int32_t member1_in, bool member2_in) : member1{member1_in}, member2{member2_in} {}
5  };
6
7  void foo() {
8    // create a smart pointer for an Object where member1 = 281 and member2 = true
9    std::unique_ptr<int32_t> ptr = std::make_unique<Object>(281, true);
10   /* ... other code ... */
11 } // ptr goes out of scope, dynamic memory for Object automatically cleaned up
```

Another example involving a unique pointer is shown below. In this example, the `DataManager` class stores a pointer to a dynamically allocated `Calculator` object. The `Calculator` is created on the heap when the `DataManager` object is constructed and deleted automatically by the smart pointer when its owning `DataManager` object goes out of scope.

```
1  // a calculator object that can be used to calculate something
2  class Calculator {
3  public:
4    int32_t calculate();
5  };
6
7  class DataManager {
8  private:
9    // a unique pointer that stores a dynamically allocated Calculator object - the smart pointer
10   // will automatically delete the Calculator from the heap after the DataManager goes out of scope
11   std::unique_ptr<Calculator> calculator;
12
13 public:
14   // DataManager constructor, use std::make_unique() to construct the internal calculator
15   DataManager(/* ... constructor args ... */) :
16     calculator{std::make_unique<Calculator>(/* ... calculator constructor args ... */)} {}
17
18   // the smart pointer behaves just like a normal pointer, so you can use pointer operators on it
19   int32_t calculate() { return calculator->calculate(); }
20 };
```

[2]For most cases, that is. There are situations where initializing from `new` may be required, but they are less common edge cases that we won't be discussing here.

Some additional member functions provided by unique pointers are summarized below.

```
template <typename T>
T* std::unique_ptr<T>::get() const noexcept;
```
Returns a pointer to the unique pointer's managed object, or `nullptr` if no object is owned.

```
{
  auto ptr = std::make_unique<int32_t>(281);
  int32_t* raw_ptr = ptr.get();
  std::cout << *raw_ptr << '\n';  // prints 281
} // raw_ptr is still managed internally by the smart pointer, so it gets cleaned up here
```

```
template <typename T>
T* std::unique_ptr<T>::release() noexcept;
```
Releases ownership of the unique pointer's managed object, if any. After calling `release()`, the unique pointer's internal pointer becomes `nullptr`, and the user who called `release()` is now responsible for deleting the memory of the released object.

```
{
  auto ptr = std::make_unique<int32_t>(281);
  int32_t* raw_ptr = ptr.release();  // the smart pointer no longer owns the pointer!
  std::cout << *raw_ptr << '\n';     // prints 281
  delete raw_ptr;                    // the code that called release() must clean up
}
```

```
template <typename T>
void std::unique_ptr<T>::reset() noexcept;
```
Relinquishes ownership of the unique pointer's managed object. The previously managed object is deleted (if non-empty), and the unique pointer's internal pointer now becomes `nullptr`.

```
template <typename T>
void std::unique_ptr<T>::reset(T* ptr) noexcept;
```
Replaces the currently managed object with another object in dynamic memory pointed to by `ptr`. The previously managed object is deleted (if non-empty).

```
{
  auto ptr1 = std::make_unique<int32_t>(281);
  auto ptr2 = std::make_unique<int32_t>(370);
  ptr1.reset(new int32_t{183});  // ptr1 now owns 183, previous memory for 281 cleaned up
  ptr1.reset(ptr2.release());    // ptr1 now owns 370, ptr2 now owns nothing (due to release())
  ptr1.reset();                  // ptr1 no longer owns anything
}
```

※ 27.4.2 *Transferring Ownership of a Unique Pointer* (✻)

One important restriction of unique pointers is that they must be used under exclusive ownership. This is because a unique pointer automatically deletes its managed object once it goes out of scope, so if multiple entities shared ownership of the same unique pointer, the unique pointer's data would be deleted multiple times (since the same unique pointer would go out of scope multiple times). Because of this, copy construction and copy assignment are both *disabled* for a `std::unique_ptr<>`.

```
{
  std::unique_ptr<int32_t> ptr1 = std::make_unique<int32_t>(281);
  std::unique_ptr<int32_t> ptr2{ptr1};  // ERROR: copy construction not allowed
  std::unique_ptr<int32_t> ptr3;        // empty unique pointer with nullptr (okay)
  ptr3 = ptr1;                          // ERROR: copy assignment not allowed
}
```

If you want a new entity to take control of a unique pointer, you must *transfer* its ownership by using either `release()` or `reset()`, or by moving the smart pointer using move semantics (i.e., with the move constructor or move assignment operator).[3] Calling `std::move()` on a unique pointer is equivalent to calling `release()` and then `reset()` on that pointer. After transferring ownership, the unique pointer that was moved from would no longer own anything (and would be set to `nullptr`). An example is shown below:

```
{
  std::unique_ptr<int32_t> ptr1 = std::make_unique<int32_t>(281);
  // ptr1 gives up ownership of 281 to ptr2
  std::unique_ptr<int32_t> ptr2 = std::move(ptr1);

  std::unique_ptr<int32_t> ptr3 = std::make_unique<int32_t>(370);
  // ownership of ptr3's 370 is assigned to ptr2, ptr3 no longer owns anything
  ptr2 = std::move(ptr3);

  // ownership of ptr2's 370 is assigned to ptr4, ptr2 no longer owns anything
  std::unique_ptr<int32_t> ptr4{std::move(ptr2)};
}
```

[3] Move semantics (including lvalues and rvalues) were covered in section 6.9, so you can look there for a more in-depth discussion on how they work.

※ 27.4.3 *Using Unique Pointers in Functions (✻)*

Since function return values are rvalues that can be moved using the move constructor or move assignment operator without making a copy, it is perfectly appropriate to return a unique pointer from a function. If you return a unique pointer from a function and assign it to another unique pointer, then the ownership of that object is transferred out of the function and into the assigned unique pointer. An example is shown below:

```
std::unique_ptr<int32_t> create_unique_ptr(int32_t num) {
  std::unique_ptr<int32_t> func_ptr = std::make_unique<int32_t>(num);
  return func_ptr;  // func_ptr gives up ownership to the object that claims the return value
} // create_unique_ptr()

void foo() {
  std::unique_ptr<int32_t> ptr1{create_unique_ptr(281)};  // ptr1 claims ownership of returned value

  std::unique_ptr<int32_t> ptr2;
  ptr2 = create_unique_ptr(370);  // ptr2 claims ownership of returned value
} // foo()
```

You may also encounter situations where you want to pass a unique pointer into a function. If you want a function to take ownership of the unique pointer being passed in, you should pass the unique pointer *by value* and call `std::move()` when passing the pointer in.

```
void foo(std::unique_ptr<int32_t> ptr);  // the foo() function takes ownership of the passed in ptr

int main() {
  std::unique_ptr<int32_t> ptr = std::make_unique<int32_t>(281);
  // pass the pointer into the function using std::move()
  // (otherwise, you would be trying to make a copy of the unique pointer, which is disallowed)
  foo(std::move(ptr));
} // main()
```

This convention is something you may see if you want to construct a class object that stores a unique pointer as a member. If the unique pointer associated with this member variable was constructed before an instance of this class object was created, you will have to transfer it into the class object's constructor using `std::move()`, as shown:

```
class DataManager {
private:
  std::unique_ptr<Calculator> calculator;  // private member variable
public:
  // constructor takes in a calculator constructed elsewhere
  // the DataManager takes ownership of the calculator that is passed in
  DataManager(std::unique_ptr<Calculator> calculator_in) : calculator{std::move(calculator_in)} {}
};

int main() {
  std::unique_ptr<Calculator> calc = std::make_unique<Calculator>();
  // pass the Calculator into the constructor of the DataManager using std::move()
  DataManager dm{std::move(calc)};
} // main()
```

There are also cases where you may need to pass a unique pointer by reference. The best example of this is if you have a function that might modify or reset the contents of the unique pointer without assuming ownership of the pointer. When passing a unique pointer by reference, you do not need to explicitly call `std::move()` (since passing by reference does not invoke the copy constructor or copy assignment operator).

```
// function that may update or reset the passed in pointer, so pass by reference
void update_ptr(std::unique_ptr<int32_t>& ptr);

int main() {
  std::unique_ptr<int32_t> ptr = std::make_unique<int32_t>(281);
  update_ptr(ptr);  // pass by reference, so no need to explicitly call std::move()
} // main()
```

Lastly, it should be mentioned that, while it is possible to pass a unique pointer by const reference, this is considered to be bad style. This is because a function that accepts a const reference to a unique pointer can neither manipulate it nor take ownership, so it thereby does not need to care about how this data is managed. Since the function only needs to view the data stored by the pointer without making any changes, it is better to pass in the raw pointer instead — this can be done using the `get()` member function of a unique pointer.

```
// not ideal, you do not need to pass the entire unique_ptr if it cannot be manipulated
void foo(const std::unique_ptr<int32_t>& ptr);

// this is a better way to pass in the contents of the pointer
void bar(const int32_t* ptr);

int main() {
  std::unique_ptr<int32_t> ptr = std::make_unique<int32_t>(281);
  bar(ptr.get());  // pass in raw pointer, which you can retrieve using get()
} // main()
```

※ 27.4.4 *Storing Unique Pointers in Containers (✻)*

Even though they cannot be copied, unique pointers can in fact be stored in STL containers thanks to move semantics. This can be done by filling the container with *rvalue* objects so that they be moved into the container (this can be done by either using an unnamed `std::unique_ptr<>` such as a function return value, or by calling `std::move()` on an existing unique pointer object). An example is shown below:

```
std::vector<std::unique_ptr<int32_t>> unique_ptr_vec;

// to insert a unique pointer, you must pass in an rvalue, as shown below
// "std::make_unique<int32_t>(281)" is an rvalue since it is a function return value
unique_ptr_vec.push_back(std::make_unique<int32_t>(281));

// if a unique pointer already exists, you must call std::move() to add it to the container
// recall that std::move() can be used to cast an lvalue to an rvalue
// after calling std::move() on ptr, ptr will no longer own anything
std::unique_ptr<int32_t> ptr = std::make_unique<int32_t>(281);
unique_ptr_vec.push_back(std::move(ptr));
```

If we store unique pointers in a container, then the ownership of those pointers also belongs to the container. If you erase a unique pointer from the container, it is destroyed and its memory is cleaned up. Similarly, once the container gets emptied or goes out of scope, all of its unique pointers will also be destroyed and cleaned up.

```
std::vector<std::unique_ptr<int32_t>> unique_ptr_vec;
unique_ptr_vec.push_back(std::make_unique<int32_t>(281));
unique_ptr_vec.push_back(std::make_unique<int32_t>(370));
unique_ptr_vec.push_back(std::make_unique<int32_t>(376));

// erases value at index 1, so the unique pointer that manages 370 is destroyed
// and its memory is cleaned up (index 0 is now 281 and index 1 is now 376 after the erase)
unique_ptr_vec.erase(unique_ptr_vec.begin() + 1);

for (const auto& ptr : unique_ptr_vec) {
  cout << *ptr << ' ';  // prints "281 376 "
} // for ptr

// clears the entire vector - all of its unique pointers are destroyed, with their memory deleted
unique_ptr_vec.clear();
```

To remove ownership of a unique pointer from the container, you will have to move the pointer out of the container. If you transfer ownership of a unique pointer out of a container without erasing it from the container, then the empty unique pointer would remain in the container. Because of this, you may need to check if a unique pointer in the container is empty before using it (luckily, unique pointers can be implicitly converted to a Boolean when used in a conditional statement, so you can check if a unique pointer has a value in an `if` statement, as shown below).

```
{ // begin a new scope
  std::vector<std::unique_ptr<int32_t>> unique_ptr_vec;
  unique_ptr_vec.push_back(std::make_unique<int32_t>(281));
  unique_ptr_vec.push_back(std::make_unique<int32_t>(370));
  unique_ptr_vec.push_back(std::make_unique<int32_t>(376));

  // move ownership of the pointer at index 1 out of the container
  // now, ptr is in charge of managing the dynamic memory for 370 (the value at index 1)
  std::unique_ptr<int32_t> ptr = std::move(unique_ptr_vec[1]);

  // after being moved out, index 1 of the vector stores an empty unique pointer
  // you will need to check for this when trying to use values in the container
  for (const std::unique_ptr<int32_t>& ptr_in_vec : unique_ptr_vec) {
    if (ptr_in_vec) {  // checks if ptr has value
      std::cout << *ptr_in_vec << '\n';
    } // if
    else {
      std::cout << "ptr is empty!\n";
    } // else
  } // for ptr_in_vec

  // vector of pointers is cleared, so all of the smart pointers it owns are cleaned up; however,
  // since 370 was transferred out of the container earlier, its memory would still be valid here
  unique_ptr_vec.clear();

  /* ... additional code ... */

} // ptr goes out of scope here, so the memory for 370 gets deleted
```

The output of the above code is:

```
281
ptr is empty!
376
```

※ 27.4.5 *Const Smart Pointers* (⁎)

Additionally, it is meaningful to discuss how `const` can be used with unique pointers (and other smart pointers in the C++ `<memory>` library). There are two places that `const` can be placed when declaring a unique pointer that manages an object of type `T`:

- **`const`** `std::unique_ptr<T>`
- `std::unique_ptr<`**`const`** `T>`

In the first declaration of `const`, the pointer itself is `const` and not what it points to. In other words, you cannot change what the pointer points to, but you can change the value it holds.

```
const std::unique_ptr<int32_t> ptr = std::make_unique<int32_t>(281);
ptr.reset();                           // ERROR: you cannot change what ptr points to
ptr = std::make_unique<int32_t>(370);  // ERROR: you cannot change what ptr points to
*ptr = 376;                            // OK: the data itself is not const
```

In the second declaration of `const`, the value that the pointer points to is `const`, but the pointer itself is not. That is, you cannot change the value that the pointer holds, but you can change what the pointer points to.

```
std::unique_ptr<const int32_t> ptr = std::make_unique<int32_t>(281);
ptr.reset();                           // OK: pointer not const and can point to something else
ptr = std::make_unique<int32_t>(370);  // OK: pointer not const and can point to something else
*ptr = 376;                            // ERROR: cannot change the value itself since it is const
```

※ 27.4.6 *Smart Pointers and Polymorphism* (⁎)

Much like normal pointers, a `std::unique_ptr<>` to a base class can be used to refer to an object of a derived type that publicly inherits from it. An example is shown below:

```
class Shape { ... };                    // base class
class Circle : public Shape { ... };    // derived class
class Triangle : public Shape { ... };  // derived class
class Rectangle : public Shape { ... }; // derived class

// OK: can use derived class to construct pointer to base class type (polymorphism)
std::unique_ptr<Shape> circle_ptr = std::make_unique<Circle>();
std::unique_ptr<Shape> triangle_ptr = std::make_unique<Triangle>();
std::unique_ptr<Shape> rectangle_ptr = std::make_unique<Rectangle>();

// these derived types can also be stored in a vector of pointers to the base type
std::vector<std::unique_ptr<Shape>> shape_vec;
shape_vec.push_back(std::move(circle_ptr));
shape_vec.push_back(std::move(triangle_ptr));
shape_vec.push_back(std::move(rectangle_ptr));
```

Because of this, unique pointers are useful if you want to create a factory function that can return multiple different types that inherit from the same base class while also ensuring that the object that is returned is properly cleaned up after it is no longer needed. A simple example of a factory function is shown below:

```
std::unique_ptr<Shape> get_shape(int32_t num_sides) {
  if (num_sides == 0) {
    return std::make_unique<Circle>();
  } // if
  if (num_sides == 3) {
    return std::make_unique<Triangle>();
  } // if
  if (num_sides == 4) {
    return std::make_unique<Rectangle>();
  } // if
  if (num_sides == 5) {
  /* ... additional if checks for different shapes ... */
  } // if

  throw std::invalid_argument{"Number of sides currently not supported"};
} // get_shape()

void foo() {
  std::unique_ptr<Shape> circle_ptr = get_shape(0);
  std::unique_ptr<Shape> triangle_ptr = get_shape(3);
  std::unique_ptr<Shape> rectangle_ptr = get_shape(4);
} // foo
```

27.5 Shared Pointers (⁎)

※ 27.5.1 *std::shared_ptr* (⁎)

When working with dynamic memory, there is a good chance that a unique pointer is good enough for your needs. The exception, however, is if you want to manage a resource that may be owned by multiple entities at the same time. In such a case, a `std::unique_ptr<>` would not work. Instead, the C++ `<memory>` library provides another type of smart pointer that can handle shared ownership: the `std::shared_ptr<>`.

An object in dynamic memory can be managed by multiple shared pointers, and it will only be destructed if there are no longer any shared pointers that reference that object. As discussed earlier, a shared pointer can tell if it's the last one pointing to an object via a technique known as *reference counting*. When a shared pointer is constructed, the reference count of its pointed to object is incremented; when a shared pointer is destructed, the reference count of its pointed to object is decremented. Once the reference count reaches zero, the shared object is deleted.

Since an object under shared ownership can be referenced by multiple shared pointers, its copy constructor and copy assignment operator are *not* disabled. If a shared pointer is copied, the reference count of its managed object is incremented. Similarly, if one shared pointer is assigned to another shared pointer, the object being assigned has its reference count incremented, and the object being overwritten has its reference count decremented. Like unique pointers, a shared pointer can be initialized directly using a raw pointer to dynamic memory, but the preferred method in almost all situations is to initialize a shared pointer using a separate factory method, `std::make_shared()`.

```
template <typename T, typename... Args>
std::shared_ptr<T> std::make_shared(Args&&... args);
```
Constructs an object of type `T` using its constructor arguments `args` and wraps it in a `std::shared_ptr<>`.

Similar to unique pointers, you can use `get()` to get the internal raw pointer that is managed by a shared pointer.

```
template <typename T>
T* std::shared_ptr<T>::get() const noexcept;
```
Returns a pointer to the shared pointer's managed object, or `nullptr` if no object is owned.

Shared pointers also support a `reset()` method, but they do *not* support a `release()` method.

```
template <typename T>
void std::shared_ptr<T>::reset() noexcept;
```
Relinquishes ownership of the shared pointer's managed object. The previously managed object is deleted if no other shared pointer owns it, and the shared pointer's internal pointer now becomes `nullptr`.

```
template <typename T>
void std::shared_ptr<T>::reset(T* ptr) noexcept;
```
Replaces the currently managed object with another object in dynamic memory pointed to by `ptr`. The previously managed object is deleted if nothing else owns it. If `ptr` is already owned by another smart pointer, the function results in undefined behavior.

Many of the features we discussed for unique pointers (such as dereferencing, storing polymorphic classes, applying `const`, passing into functions, storing in containers, etc.) also apply to shared pointers. The main difference is that shared pointers can be copied, so you do not need to explicitly call `std::move()` on them if you want to construct another owner of the pointer — a simple assignment or copy would suffice.

```
1  int main() {
2    // new shared pointer, ref count for 281 is 1
3    std::shared_ptr<int32_t> ptr1 = std::make_shared<int32_t>(281);
4
5    { // begin a new scope
6      // ptr1 and ptr2 share the same object, ref count for 281 now 2
7      std::shared_ptr<int32_t> ptr2 = ptr1;
8      std::cout << *ptr2 << '\n';  // prints 281
9    } // ptr2 goes out of scope, so ref count of 281 goes back to 1
10
11   // shared pointers can also be stored in containers, no moving necessary due to shared ownership
12   std::vector<std::shared_ptr<int32_t>> shared_ptr_vec;
13   // shared pointer of 281 is copied, ref count goes to 2
14   shared_ptr_vec.push_back(ptr1);
15   shared_ptr_vec.push_back(std::make_shared<int32_t>(370));
16
17   // this loop prints "281 370 "
18   for (const auto& shared_ptr : shared_ptr_vec) {
19     if (shared_ptr) {
20       std::cout << *shared_ptr << ' ';
21     } // if
22   } // for shared_ptr
23
24   // vector is cleared; nothing owns 370 anymore, so it gets deleted
25   // ptr1 still owns 281, and ref count drops from 2 to 1
26   shared_ptr_vec.clear();
27
28   std::cout << *ptr1 << '\n';  // prints 281
29 } // ptr1 goes out of scope; it is the last owner of 281, so 281 gets deleted
```

Remark: Even though two shared pointers can reference the same object, you should *not* initialize two shared pointers with the same raw pointer! The following code would cause a double deletion (which is undefined behavior):

```
int main() {
  int32_t* val = new int32_t{281};
  std::shared_ptr<int32_t> ptr1{val};
  std::shared_ptr<int32_t> ptr2{val};  // BAD: causes double delete
} // main()
```

We will discuss why shortly, but this is related to how shared pointers manage their reference counts behind the scenes.

Shared pointers also provide a `use_count()` member method, which can be used to return the number of shared pointers that are referencing the current object. If a shared pointer has no managed object, calling `use_count()` on it returns zero.

```
template <typename T>
long std::shared_ptr<T>::use_count() const noexcept;
```

Returns the number of different shared pointers that are referencing the current object, and zero if there is no managed object. This method is approximate in multithreaded environments.

※ 27.5.2 *Converting Between Unique and Shared Pointers* (✻)

A unique pointer (exclusive ownership) can be easily converted to a shared pointer (shared ownership). When this happens, the shared pointer takes ownership of the unique pointer that it is assigned to. This makes a unique pointer a good return type for a factory function, since this allows the caller of the function decide whether they want the returned object to be under exclusive or shared ownership.

```
std::unique_ptr<int32_t> foo() {
  return std::make_unique<int32_t>(280);
} // foo()

int main() {
  std::unique_ptr<int32_t> u_ptr1 = std::make_unique<int32_t>(281);
  std::shared_ptr<int32_t> s_ptr1 = std::move(u_ptr1);                // OK
  std::shared_ptr<int32_t> s_ptr2 = std::make_unique<int32_t>(370);   // OK
  std::shared_ptr<int32_t> s_ptr3 = foo();                            // OK
} // main()
```

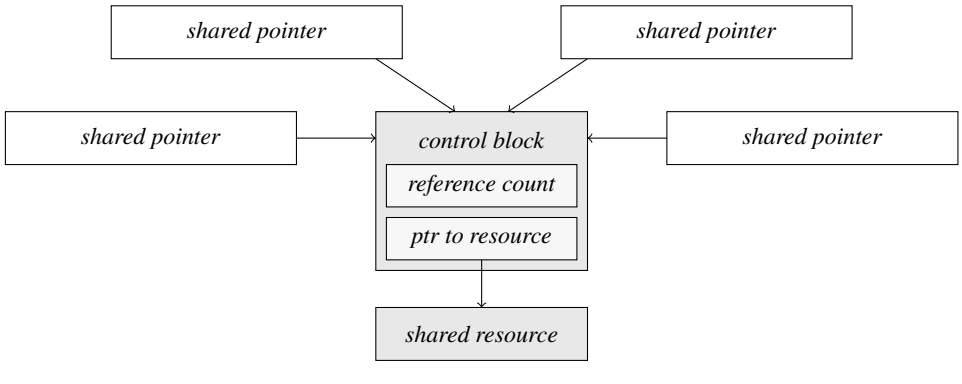
Converting the other way, however, is not possible. Once you place an object under shared ownership, you can no longer manage it with a unique pointer, even if its reference count is one.

```
int main() {
  std::unique_ptr<int32_t> u_ptr1 = std::make_shared<int32_t>(281);  // ERROR
  std::shared_ptr<int32_t> s_ptr1 = std::make_unique<int32_t>(370);
  std::unique_ptr<int32_t> u_ptr2 = std::move(s_ptr1);               // ERROR
} // main()
```

This rule makes sense: if you have a resource that you know is only owned by one entity (e.g., a unique pointer), then it is perfectly safe to allow other entities to share ownership of that resource without any conflicts. However, once multiple entities are allowed to share the resource, it is no longer safe to restrict its access back to a single entity, since you would have to take away ownership from all other users of the resource. Because of this, it is good practice to consider if a unique pointer is sufficient before you reach for a shared pointer: not only are unique pointers more lightweight, but an overreliance on shared pointers makes it harder to go back to a unique pointer when exclusive ownership is expected.

※ 27.5.3 *Shared Pointers and Reference Counting* (✻)

Unlike unique pointers, shared pointers are roughly twice the size of a raw pointer, since they need to store additional information for reference counting. For a `std::shared_ptr<>`, reference counting is done through the non-intrusive control block approach. If you recall from earlier, this approach involves a dynamically allocated control block object that keeps track of a shared resource's reference count. Each shared pointer that references the shared resource points to this control block.



A control block needs to be created when the first shared pointer is created for a shared resource. The following rules are used to determine when a control block is created:

1. A control block is created if `std::make_shared()` is used to create a shared pointer. Since this method returns a shared pointer to a newly constructed object in dynamic memory, the returned shared pointer is guaranteed to be the first one referencing this object.

   ```
   // creates a new control block
   std::shared_ptr<int32_t> ptr = std::make_shared<int32_t>(281);
   ```

2. A control block is created if a `std::shared_ptr<>` is constructed from a `std::unique_ptr<>`. An object managed by a unique pointer does not have a control block (since it can only be managed by one owner), so transferring the object to a shared pointer means that a control block must be created.

   ```
   // creates a new control block
   std::unique_ptr<int32_t> u_ptr = std::make_unique<int32_t>(281);
   std::shared_ptr<int32_t> s_ptr = std::move(u_ptr);
   ```

3. A control block is created if a `std::shared_ptr<>`'s constructor is called with a raw pointer. If you construct a shared pointer directly with a raw pointer, it assumes that it is the first smart pointer that references that raw pointer. *This is why you cannot initialize two shared pointers with the same raw pointer!* Otherwise, you would get two control blocks (and thus two reference counts) for the same object; this leads to a double delete since both reference counts would eventually reach zero. Instead, if a shared resource is already managed by other shared pointers, you should create a new shared pointer via a method that does not create a new control block.

   ```
   // creates a new control block
   std::shared_ptr<int32_t> ptr{new int32_t{281}};
   ```

If a shared pointer is copied or assigned from another shared pointer, then a control block is *not* created. This is because the old shared pointer being copied or assigned from must already have a control block of its own. As a result, a shared pointer created via a copy or assignment simply increments the reference count of this existing control block.

Similarly, when a shared pointer is destroyed or reassigned, it decrements the reference count in its control block. When a shared pointer decrements its reference count to zero, it deletes the managed object as well as its corresponding control block.

※ 27.5.4 *std::enable_shared_from_this* (✻)

One particularly interesting edge case involves shared pointers that return `this`. Using the rules for control blocks, we know that the following code causes undefined behavior, since two reference counts would be created for the same object:

```
int32_t* ptr = new int32_t{281};
std::shared_ptr<int32_t> s_ptr1{ptr};
std::shared_ptr<int32_t> s_ptr2{ptr};
```

However, there may be cases where you will need to return a `std::shared_ptr<>` that owns the object it is called on. Upon first glance, it may seem appropriate to implement this behavior as follows:

```
class Thing {
public:
  std::shared_ptr<Thing> get_pointer_to_self() {
    return std::shared_ptr<Thing>(this);  // "this" is a pointer to the current Thing instance
  } // get_pointer_to_self()
};
```

Unfortunately, this code is incorrect and may cause undefined behavior. To see why, consider the following code that uses a `Thing` object:

```
void foo() {
  std::shared_ptr<Thing> s_ptr1{new Thing{}};
  std::shared_ptr<Thing> s_ptr2 = s_ptr1->get_pointer_to_self();
} // foo()
```

When `shrd_ptr1` is constructed, it is the first one to own the newly allocated `Thing` instance, so it creates a control block. However, when `shrd_ptr2` is constructed, it also creates a control block for the exact same object, as it is unaware that another shared pointer is already managing the current `Thing`. This leads to undefined behavior via a double delete, since the `Thing` would have its memory deleted twice!

To account for this edge case, you can inherit the class template `std::enable_shared_from_this<>`, which takes in the type of the object that needs to return a pointer to itself. The `std::enable_shared_from_this<>` class implements a special member function called `shared_from_this()`, which allows an object to return a shared pointer to itself without duplicating control blocks. The previously code is correctly implemented using `shared_from_this()` below:

```
class Thing : public std::enable_shared_from_this<Thing> {
public:
  std::shared_ptr<Thing> get_pointer_to_self() {
    return shared_from_this();  // returns a shared_ptr to self without duplicating control blocks
  } // get_pointer_to_self()
};

void foo() {
  std::shared_ptr<Thing> s_ptr1{new Thing{}};
  std::shared_ptr<Thing> s_ptr2 = s_ptr1->get_pointer_to_self();  // this is safe now
} // foo()
```

It is important to note that `shared_from_this()` is only safe to use if there exists another shared pointer that already points to `this`. If no shared pointer is currently managing the current object, invoking `shared_from_this()` would result in undefined behavior prior to C++17, and a `std::bad_weak_ptr` exception since C++17. The following code would not work:

```
class Thing : public std::enable_shared_from_this<Thing> {
public:
  std::shared_ptr<Thing> get_pointer_to_self() {
    return shared_from_this();
  } // get_pointer_to_self()
};

void foo() {
  Thing* raw_thing = new Thing{};
  std::shared_ptr<Thing> s_ptr = raw_thing->get_pointer_to_self();  // undef behavior or exception
} // foo()
```

To prevent this from happening, objects that inherit `std::enable_shared_from_this<>` often hide their constructors as `private` members to prevent a user from creating an instance of the class that is not managed by a shared pointer. Instead, separate factory functions are provided to the user that always return a shared pointer to the object.

```
class Thing : public std::enable_shared_from_this<Thing> {
public:
  // separate factory function that returns a shared pointer to a new Thing -
  // a user must use this to construct a Thing to prevent get_pointer_to_self()
  // from being incorrectly called on an instance that is not already owned by a shared_ptr
  static std::shared_ptr<Thing> create();

  std::shared_ptr<Thing> get_pointer_to_self() {
    return shared_from_this();
  } // get_pointer_to_self()

private:
  // Thing constructors are declared here
};

void foo() {
  std::shared_ptr<Thing> s_ptr1 = Thing::create();
  std::shared_ptr<Thing> s_ptr2 = s_ptr1->get_pointer_to_self();
} // foo()
```

27.6 Weak Pointers (⁎)

※ 27.6.1 *The Cycle Problem* (⁎)

There is still one major issue involving shared pointers that we have yet to address, known as the *cycle problem*. To demonstrate this issue, consider the following example involving `Student` objects. Here, each `Student` can be paired up with another `Student` for a homework assignment, where a student's partner is stored internally using a shared pointer.

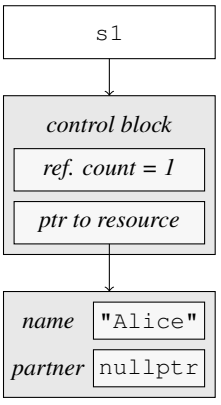
```
class Student {
private:
  std::string name;
  std::shared_ptr<Student> partner;
public:
  Student(const std::string& name_in) : name{name_in} {}

  // sets the partner of two pairs of students
  // NOTE: this is a friend function, so the function is not a class method but can
  // access private member variables of the Student class
  friend bool set_partners(const std::shared_ptr<Student>& s1, const std::shared_ptr<Student>& s2) {
    if (!s1 || !s2 || s1 == s2) {
      return false;
    } // if
    s1->partner = s2;
    s2->partner = s1;
    return true;
  } // set_partners()
};
```

Consider the following code, which creates two students and partners them together. Since these students are managed by shared pointers, it may seem reasonable to assume that both students will be cleaned up after `foo()` terminates.

```
void foo() {
  std::shared_ptr<Student> s1 = std::make_shared<Student>("Alice");
  std::shared_ptr<Student> s2 = std::make_shared<Student>("Bob");
  set_partners(s1, s2);
} // foo()
```

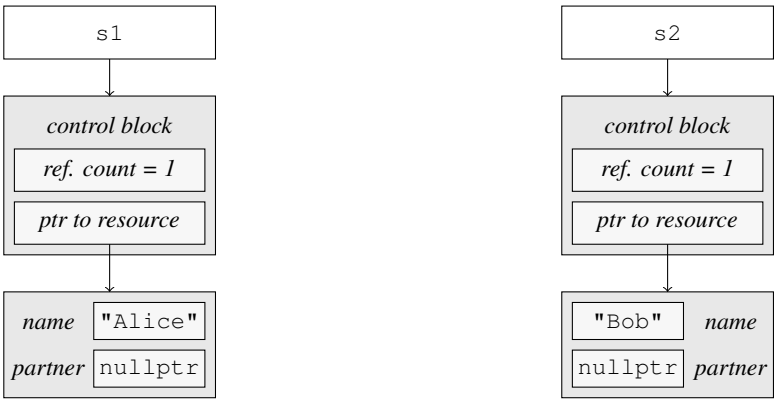
However, this is not the case! Instead, both students end up getting leaked. To understand why this happens, consider what actually happens behind the hood with these shared pointers. When `s1` is created, a shared pointer and control block are created for the student `"Alice"`.

```
std::shared_ptr<Student> s1 = std::make_shared<Student>("Alice");
```



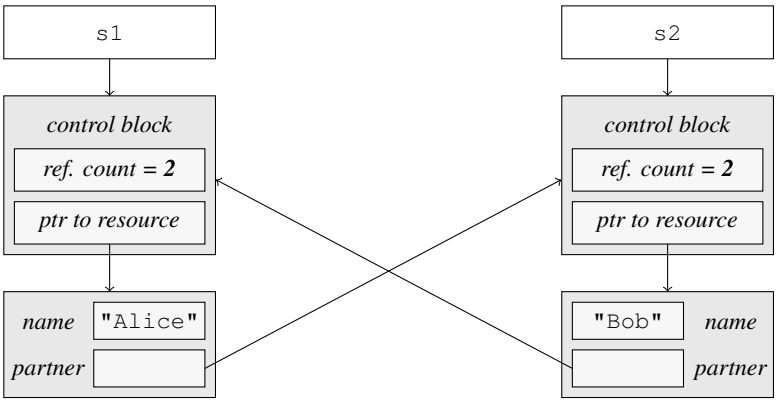
Similarly, when `s2` is created, a shared pointer and control block are created for the student `"Bob"`.

```
std::shared_ptr<Student> s2 = std::make_shared<Student>("Bob");
```



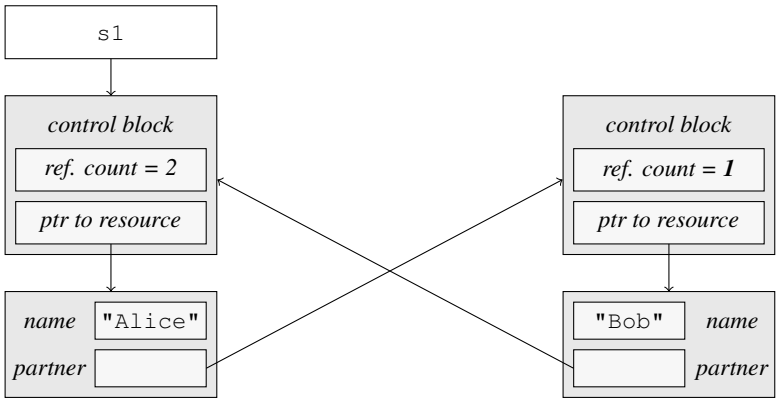
The interesting behavior happens when we call `set_partners()`. During this function call, `s1`'s partner is set to `s2`, and `s2`'s partner is set to `s1`. This increments the reference counts of both students by one, as shown:

```
s1->partner = s2;
s2->partner = s1;
```



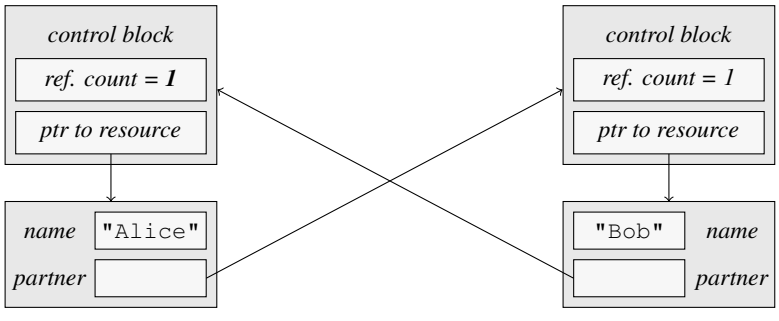
Next, the `foo()` function runs to completion, and the `s1` and `s2` shared pointers go out of scope. Since destructors are run in LIFO order at the end of a scope, `s2` goes out of scope first. `s2`'s destructor checks if anything else is still referencing `"Bob"` (i.e., is the reference count zero?). In this case, there is: the `partner` member of `"Alice"`. Because of this, the memory for `"Bob"` is not immediately deallocated.

`s2` *goes out of scope*



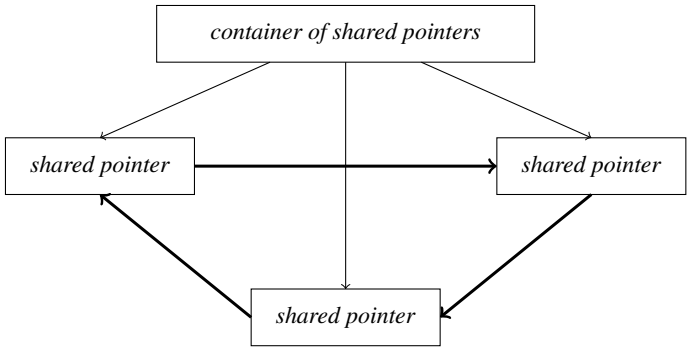
Next, `s1` goes out of scope. `s1`'s destructor checks if anything else is still referencing `"Alice"`. Since the `Student` object associated with `"Bob"` has not yet been deallocated, there is, so the memory for `"Alice"` is not immediately deallocated either.

`s1` *goes out of scope*

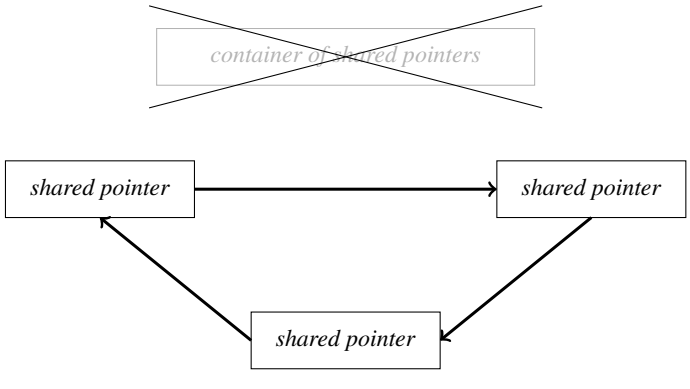


We have a problem! Even though `s1` and `s2` are both out of scope, the memory allocated for their students have not been deallocated. Since this memory is no longer accessible, we now have a memory leak.

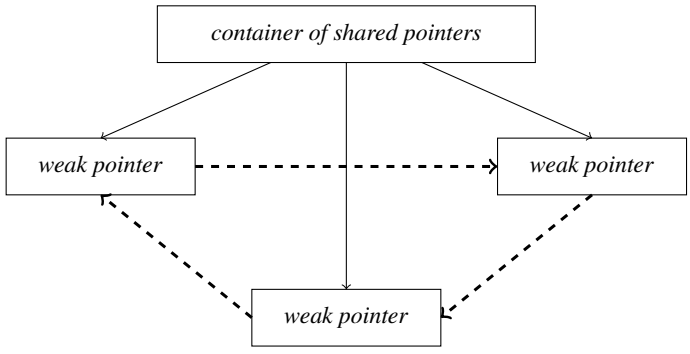
This is the essence of the cycle problem: if you have multiple shared pointers that point to each other in the form of a circular reference (e.g., $A \rightarrow B \rightarrow C \rightarrow A$), then those objects will not get properly cleaned up since they would keep each other alive.



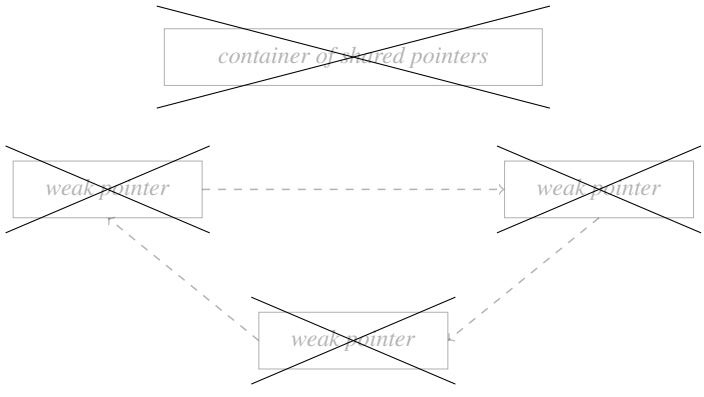
container goes out of scope, but the shared pointers keep each other alive (since the reference count of each object never drops to zero)



To fix this issue, we will need a new type of smart pointer that can be used to *observe* a shared object *without* affecting its reference count, so that it does not prevent the object from being properly destructed after it is no longer used. This special type of smart pointer is known as a **weak pointer**. By using weak pointers to point to objects in a cycle instead of normal shared pointers, we can ensure that those objects can be properly deallocated once they are no longer being used.

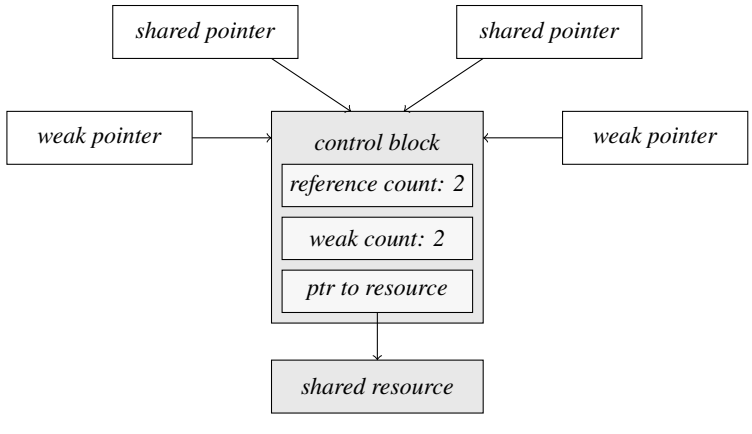


container goes out of scope, so its memory gets properly cleaned up (weak pointers do not add to reference count, so all ref counts are zero)



※ 27.6.2 *Implementing a Weak Pointer* (✻)

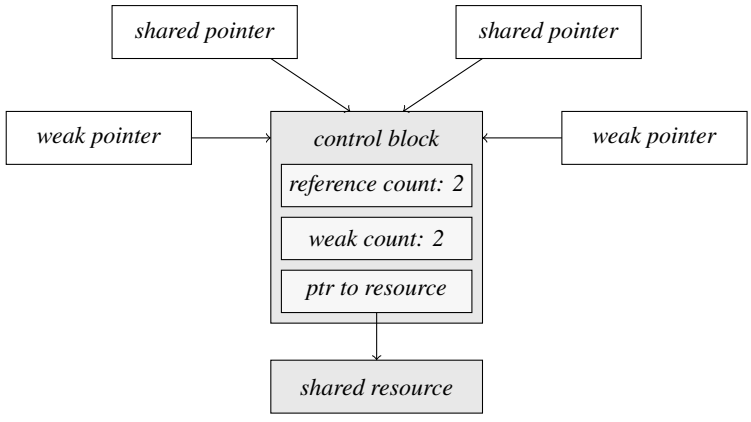
A weak pointer can be used to observe a shared object without affecting its reference count, which thereby prevents it from keeping the shared object alive when it should not be. How do we implement the functionalities of a weak pointer? It turns out that the non-intrusive control block implementation we discussed previously can be easily updated to support this new pointer type. This is done by adding a *weak count* that is maintained by weak pointers, in addition to the reference count that is maintained by shared pointers.



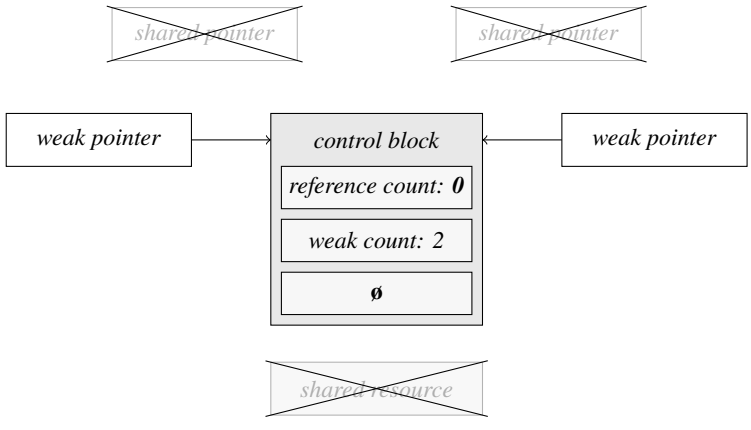
When a weak pointer is created, it points to the control block of the object it observes, and the weak count of that control block is incremented by one. Likewise, when a weak pointer is destroyed or reassigned, the weak count of its associated control block is decremented by one.

Why do we need a separate weak count in our control block? Remember that weak pointers cannot modify the reference count since they are mere observers (and not users) of the shared object they manage, so the presence of a weak pointer has no influence on when the shared object can be safely deleted (i.e., an object having a weak pointer $\neq$ something still needs that object to stay around in memory). However, this means that it is entirely possible for a weak pointer to still observe an object even after its reference count reaches zero — we do not want our weak pointer to dangle, so we cannot erase the existence of the object entirely. This is done by keeping the control block around if the weak count is not zero! That is, a shared resource is deallocated once its reference count is zero, but its corresponding control block is deallocated only after *both* its reference count and weak count are zero.

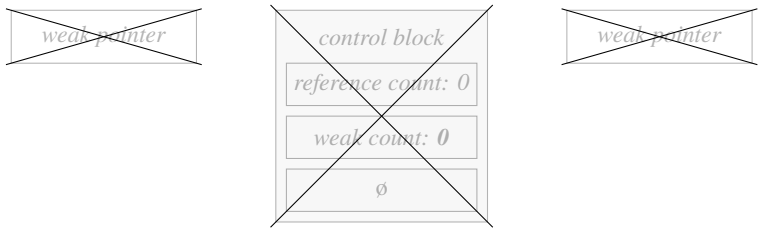
To illustrate this process, consider the following example, where a shared resource is pointed to by two shared pointers and two weak pointers. Since weak pointers do not indicate ownership and thus do not affect the reference count, the reference count is two for the number of shared pointers. The number of weak pointers is stored separately as a weak count, which also has a value of two.



Now, suppose both shared pointers go out of scope, which drops the reference count down to zero. Without any weak pointers, both the shared resource and control block would get deallocated. In this situation, however, there are still two weak pointers that point to the shared resource. We still want to delete the shared resource (since a reference count of zero indicates that nothing needs to use it anymore), but we cannot leave the weak pointers dangling. To address this, the shared resource gets deleted but the control block stays around.



Only after both weak pointers go out of scope does the weak count drop back to zero. When this happens and the reference count is *also* zero, the last weak pointer to be destroyed also cleans up the memory used by the control block.



It should be noted that, unlike unique and shared pointers, weak pointers in the C++ standard library are significantly more limited in functionality and cannot be used syntatically like a normal raw pointer. For example, you cannot dereference a weak pointer to access its underlying value, and a weak pointer must be initialized from a existing shared pointer. The only thing you can really do with a weak pointer is query whether the object it is observing still exists (which is done by looking at the reference count of the control block, which is guaranteed to stay alive as long as there are still weak pointers referencing it). That being said, a weak pointer is still more useful than a normal raw pointer because of this ability to check if its underlying object is still alive — and, in the case where the object is still alive, you can easily use the weak pointer to construct a new shared pointer that can be dereferenced to access the object's value.

※ 27.6.3 *std::weak_ptr* (✻)

The C++ <memory> library provides the `std::weak_ptr<>`, which is a class that implements the functionalities of a weak pointer that we discussed earlier. Because weak pointers are designed to observe an existing shared resource, they can only be constructed from an existing shared pointer, as shown.

```
std::shared_ptr<int32_t> s_ptr = std::make_shared<int32_t>(281);

// creates a weak_ptr that observes the memory of s_ptr without upping its reference count
std::weak_ptr<int32_t> w_ptr = s_ptr;
```

A `std::weak_ptr<>` cannot be dereferenced directly — on its own, a weak pointer can only tell you whether the object it is pointing to is alive, and not what the object's value is. To check if a weak pointer's object has been deleted or not, you can use the `expired()` member.

```
template <typename T>
bool std::weak_ptr<T>::expired();
```
Returns `true` if the object managed by the weak pointer has already been deleted, and `false` otherwise.

```
std::weak_ptr<int32_t> w_ptr;

{ // begin a new scope
  std::shared_ptr<int32_t> s_ptr = std::make_shared<int32_t>(281);
  w_ptr = s_ptr;
  std::cout << w_ptr.expired() << '\n';  // prints 0 since s_ptr's object is still alive
} // ptr goes out of scope

std::cout << w_ptr.expired() << '\n';    // prints 1 since s_ptr's object has been deleted
```

If a weak pointer is not expired, you can use the `lock()` member function to return a new `std::shared_ptr<>` to the managed object. This allows you to dereference and access the shared object's value.

```
template <typename T>
std::shared_ptr<T> std::weak_ptr<T>::lock() const noexcept;
```
Creates a new `std::shared_ptr<>` that references the shared object that the weak pointer is observing. If the weak pointer is expired, then the returned shared pointer is empty.

```
std::shared_ptr<int32_t> s_ptr = std::make_shared<int32_t>(281);
std::weak_ptr<int32_t> w_ptr = s_ptr;
if (auto access_ptr = w_ptr.lock()) {
  std::cout << *access_ptr << '\n';  // prints 281
} // if
```

Using this information, let us return to our initial `Student` example. Instead of storing each student's partner as a shared pointer (which lead to a circular reference that prevented the students from being properly deleted), we will store it as a *weak* pointer:

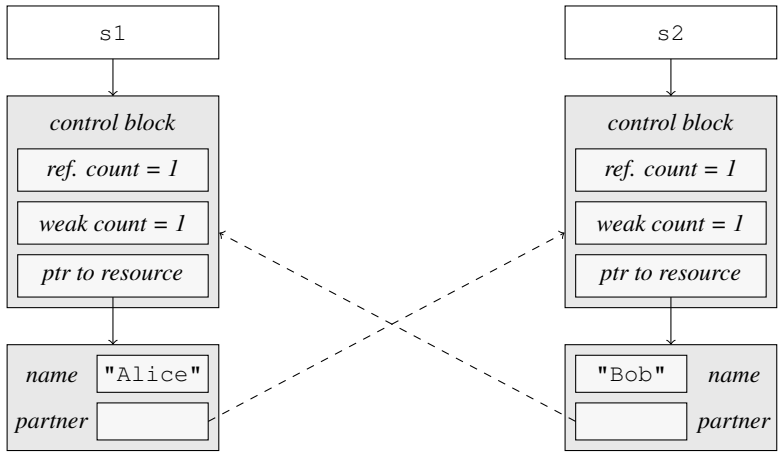
```
class Student {
private:
  std::string name;
  std::weak_ptr<Student> partner;  // WEAK pointer and not shared!
public:
  Student(const std::string& name_in) : name{name_in} {}

  // sets the partner of two pairs of students
  // NOTE: this is a friend function, so the function is not a class method but can
  // access private member variables of the Student class
  friend bool set_partners(const std::shared_ptr<Student>& s1, const std::shared_ptr<Student>& s2) {
    if (!s1 || !s2 || s1 == s2) {
      return false;
    } // if
    s1->partner = s2;
    s2->partner = s1;
    return true;
  } // set_partners()
};
```

Now, we will run the same code that previously caused a memory leak:

```
void foo() {
  std::shared_ptr<Student> s1 = std::make_shared<Student>("Alice");
  std::shared_ptr<Student> s2 = std::make_shared<Student>("Bob");
  set_partners(s1, s2);
} // foo()
```

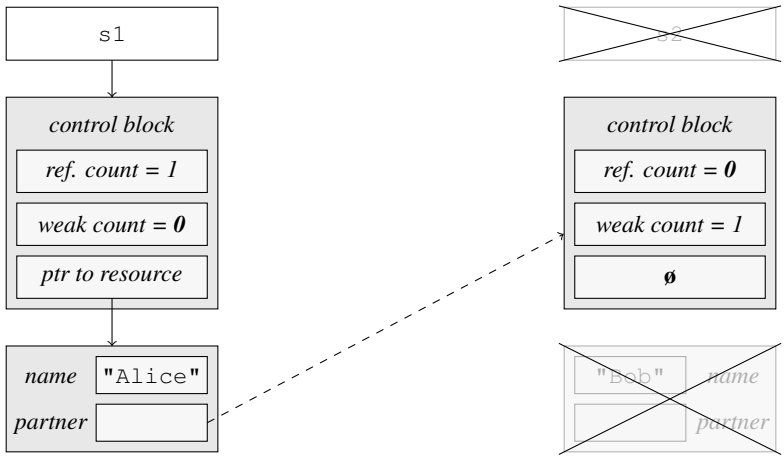
What happens now? When we assign the partners in the `set_partners()` method, the new partners do *not* affect each student's reference count since they are stored as weak pointers. Both students would still have a reference count of one after their partners are set!

```
s1->partner = s2;
s2->partner = s1;
```



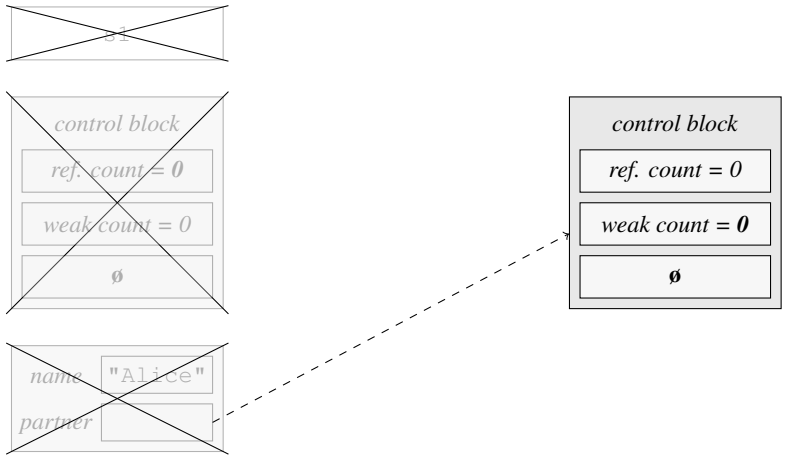
When `s2` goes out of scope, the reference count of `"Bob"` drops to zero, so the memory allocated to the student `"Bob"` is deleted. However, since the weak count is not yet zero (as `"Alice"` is still referencing `"Bob"` via its weak pointer), the control block of `"Bob"` is not deleted. Note that the weak pointer of `"Bob"` also gets destructed here, which decrements the weak count of `"Alice"`.

`s2` *goes out of scope*

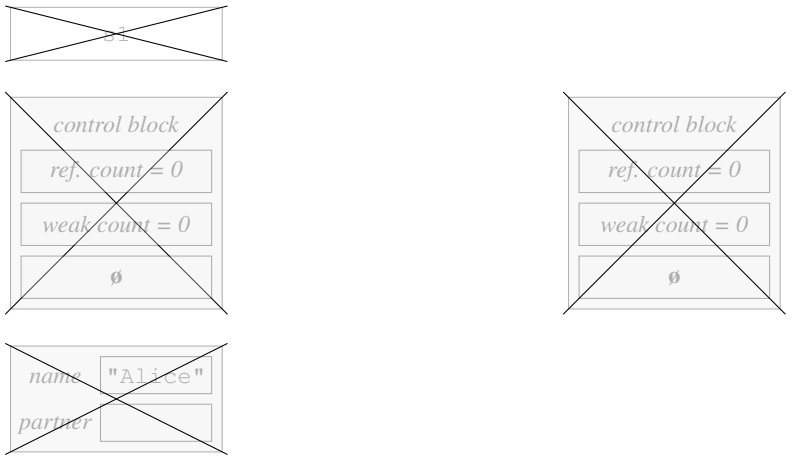


Next, `s1` goes out of scope, so the reference count of `"Alice"` is decremented to zero. Since both the reference count *and* weak count of `"Alice"` are now zero, the memory allocated to the student `"Alice"` and its control block are both cleaned up.

`s1` *goes out of scope*



While deleting `"Alice"`, its weak pointer is also destructed, which decrements the weak count of (what was once) `"Bob"`'s control block to zero. Since both the reference count and weak count of this control block are now zero, it is deleted as well.



Both students (and their control blocks) have been successfully cleaned up, so no memory is leaked.

27.7 When Should Smart Pointers Be Used? (⁎)

To summarize, the introduction of smart pointers in C++11 made it easier to manage the lifetime of dynamic memory, ensuring that an object created on the heap also gets properly cleaned up once it is out of use. There are three main categories of smart pointers provided by the C++ `<memory>` library: unique, shared, and weak pointers.

A unique pointer is the simplest type of smart pointer, designed to maintain *exclusive* ownership of an object in dynamic memory. These pointers automatically delete their managed object after it is no longer used, and they also have very little overhead compared to other types of smart pointers. As a result, unique pointers are an ideal choice for managing dynamic memory in most situations.

Unique pointers should not be used, however, if a dynamically allocated resource requires shared ownership. In that case, you can use a shared pointer instead, which uses reference counting to ensure that an object is only deleted after *all* of its owners no longer need it. For scenarios where shared resources may form a cycle, you can use a weak pointer in place of a shared pointer, as weak pointers observe a shared object without increasing its reference count (thereby allowing objects in a cycle to be properly cleaned up).

With smart pointers at hand, is there ever a case where a normal raw pointer should be used over a smart pointer? This answer is a bit more nuanced, since it depends on how the pointer is intended to be used. The core function of smart pointers revolves around the concept of dynamic memory *ownership*. An entity in charge of owning or managing the lifetime of a dynamically-allocated object should avoid raw pointers in almost all situations — if you are allocating a brand new object on the heap, you should try your best to *always* store it in a smart pointer, as they can prevent many types of memory issues and save you a lot of trouble later on.

However, if an entity needs to use the contents of a dynamically allocated object that is already owned and managed by something else, and you can ensure that the object will always be alive when you use it, then it would be okay to pass a raw pointer to that object (provided that you never delete it outside the scope in which it is owned). To illustrate, consider the following `Service` class that is responsible for starting up a service. Each instance of a `Service` maintains a logger object that can be used log messages from the service to a log file; the logger is allocated dynamically and is stored in a unique pointer:

```
1  class Service {
2  private:
3    std::unique_ptr<Logger> logger;
4  public:
5    // member functions
6  };
```

Now, suppose each `Service` also maintains a separate `DataRetriever` class that is used to retrieve data that is needed for the service to run. The lifetime of a `DataRetriever` object is restricted within the scope of the `Service` instance it resides in, so it is also stored as a member variable of the `Service` class, as shown:

```
1  class Service {
2  private:
3    std::unique_ptr<Logger> logger;
4    std::unique_ptr<DataRetriever> data_retriever;
5  public:
6    // member functions
7  };
```

Additionally, suppose the `DataRetriever` wants to log messages to the log file as well, and thus also needs a copy of the logger object that is stored in its encompassing service. How should this be done?

```
1  class DataRetriever {
2  private:
3    // needs to store the logger here!
4  public:
5    // member functions
6  };
```

One potential strategy is to store the logger as a shared pointer within *both* the `Service` class and the `DataRetriever` class. When the service is first created, the shared pointer is created normally. Then, when the data retriever is constructed within the service, the shared pointer of the service is passed into the data retriever's constructor and used to initialize its own logger.

```
class Service {
private:
  std::shared_ptr<Logger> logger;
  std::unique_ptr<DataRetriever> data_retriever;
public:
  Service(const std::shared_ptr<Logger>& logger_in) : logger{logger_in} {
    // the data retriever initializes its own logger from a shared pointer
    data_retriever = std::make_unique<DataRetriever>(logger);
  } // Service()
};

class DataRetriever {
private:
  std::shared_ptr<Logger> logger;
public:
  DataRetriever(const std::shared_ptr<Logger>& logger_in) : logger{logger_in} {}
};

int main() {
  // construct logger and pass into Service constructor
  std::shared_ptr<Logger> service_logger = std::make_shared<Logger>();
  Service new_service{service_logger};

  /* ... additional code ... */
} // main()
```

However, this is not necessary. Recall from earlier that we define the *owner* of a dynamically allocated object as the entity responsible for cleaning up that object after it is done using it. In this case, the logger really only has one owner: the `Service` class that it belongs to. The `DataRetriever` is just a component of the service that uses the logger, but it does not need to manage the lifetime of the logger itself.

Because the data retriever does not claim ownership on the logger object, and we know that the logger will stay alive during the entire lifespan of the data retriever (since they are both managed under the exact same service), it is better to store the logger as a *raw* pointer within the `DataRetriever` class. Not only does this convey clearer intent on who owns the logger, it also avoids the additional overhead of a shared pointer when storing an object that can be managed under exclusive ownership.

```
class Service {
private:
  std::unique_ptr<Logger> logger;
  std::unique_ptr<DataRetriever> data_retriever;
public:
  Service(std::unique_ptr<Logger> logger_in) : logger{std::move(logger_in)} {
    // pass in a raw pointer to construct the data retriever (which can be obtained using .get())
    data_retriever = std::make_unique<DataRetriever>(logger.get());
  } // Service()
};

class DataRetriever {
private:
  Logger* logger;
public:
  DataRetriever(Logger* logger_in) : logger{logger_in} {}
};

int main() {
  // construct logger and pass into Service constructor
  std::unique_ptr<Logger> service_logger = std::make_unique<Logger>();
  Service new_service{std::move(service_logger)};

  /* ... additional code ... */
} // main()
```

In conclusion, while there are cases where using a raw pointer may be permissible (such as the one above), it is generally good advice to use smart pointers whenever you are dealing with the *ownership* of a dynamically allocated object. When allocating dynamic memory, you should avoid creating an object that is managed explicitly by `new` and `delete`. This is because there is little downside and enormous upside in using smart pointers to automate memory management — the implementations provided by the C++ standard library are not only efficient and simple to use, but a continual reliance on smart pointers can also make your programs less susceptible to several kinds of memory-related bugs.