Hands-On Design Patterns with C++

Master the Design Patterns to create robust, reusable and easily maintainable apps

By Fedor G Pikus

Packt
www.packt.com

Table of Contents

14. Template Method Pattern and Non-Virtual Idiom
15. Curiously Recurring Template Pattern
16. Mixin
17. Scopeguard
18. Serialization/Deserialization and Virtual Template
19. Policy-Based Design
20. Policy-Based Design, Inside-Out
21. Typelist and Typemap
22. Visitor and Multiple Dispatch

Preface

Chapter 1, *An Introduction to Inheritance and Polymorphism*, This chapter summarizes the use of class inheritance in C++, focused on the aspects critical for the understanding of design pattern. This chapter explains the virtual functions and run-time dispatch. It also seeks to dispel the misconceptions or misunderstandings surrounding the use of inheritance and virtual functions. We also discuss the restrictions on polymorphic use of objects and how you might run afoul of them, with unexpected results.

Chapter 2, *Function Templates*, This chapter is not a comprehensive guide to function templates. There are dedicated books for that. Instead, we highlight the features of the template language that will come useful in the next chapters, when we start discussing the patterns. The reader will also learn the most common pitfalls and gotchas surrounding the template use.

Chapter 3, *Class Templates*, Class templates have even more complex set of rules than function templates, and this isn't a book to list them all. Fortunately, C++ is designed with the attitude that the basic understanding (as long as it's correct) should be sufficient to handle the "normal" uses of the language, and things mostly work the way you expect. This chapter starts with a simplified overview of templates that is sufficient for the absolute majority of hands-on applications. We then highlight some of the finer points and potential traps that a practicing C++ programmer might run into in his daily work.

Chapter 4, *Swap*, This chapter is dedicated to a very simple,

even humble C++ feature: swap function that swaps two objects, often two containers. This very simple feature, however, is connected to such challenging aspects of C++ as resource management and exception safety. Swap is, arguably, one of the most expressive C++ idioms: so much is effectively said, or implied, by just a few characters of code.

Chapter 5, *A Comprehensive Look at RAII*, This chapter introduces one of the most important C++ idioms, Resource Acquisition Is Initialization (RAII). It's hard to find a C++ program of more than few lines which does not use this idiom. And yet, it has several limitations or annoyances that, at best, make the code much less readable. At worst, the programmer gives up on RAII and goes on to [mis]manage the resources manually. Understanding these issues leads us toward some of the design patterns described in the latter chapters.

Chapter 6, *TBD – OOP Idiom*, This chapter introduces one of the most important C++ idioms, Resource Acquisition Is Initialization (RAII). It's hard to find a C++ program of more than few lines which does not use this idiom. And yet, it has several limitations or annoyances that, at best, make the code much less readable. At worst, the programmer gives up on RAII and goes on to [mis]manage the resources manually. Understanding these issues leads us toward some of the design patterns described in the latter chapters.

Chapter 7, *Type Erasure*, This chapter explains one of the more mysterious C++ idioms, the type erasure. Using an example from the previous chapter, we will show what exactly is being erased and where from. We demonstrate the applications of type erasure. Once the reader understands how type erasure works, he may be tempted to use it everywhere - it is that addictive. There is, however, a very good reason not to: we show how type erasure may negatively impact program's performance.

Chapter 8, *SFINAE and Overload Resolution Management*, SFINAE stands for "substitution failure is not an error" and is one of the more convoluted C++ idioms. Chances are you already used it when instantiating template functions and was not aware of it, because it quietly made the right thing happen. Still, if it ain't broken, you're not using it to its full potential, and SFINAE has a lot of potential. It can be used to manipulate the rules that determine which of the overloaded functions gets called, depending on the types and features of the arguments.

Chapter 9, *Classic Patterns for Object-Oriented Designs*, This chapter review the application and implementation of the best known patterns of object-oriented programming in C++. These are the patterns collected in the well-known book "Design Patterns: Elements of Reusable Object-Oriented Software." The reader is referred to that book for the detailed analysis of the patterns. This chapter will show an example C++ implementation and highlight any C++-specific considerations.

Chapter 10, *Local Buffer Optimization*, This is a performance optimization pattern with remarkably broad applications, from strings to type erasure. When applied correctly, it dramatically reduces the overhead of dynamic memory allocations as well as the cost of memory accesses. As an optimization pattern, it deals with design of data structures.

Chapter 11, *Friend Factory*, This pattern has nothing to do with the popular Factory pattern that is used to construct objects of different types. This factory constructs functions, and can be used to automatically generate a (non-template) function every time a class template is instantiated for a different type.

Chapter 12, *Virtual Constructors and Factories*, In C++, every member function of a class can be made virtual, including the destructor. Every function except one, that is: the constructors

are never virtual. That makes sense: virtual functions rely on knowing the object's true type at run time, and an object that is being constructed has no type yet. Nonetheless, we often need to construct objects whose type will become known only at run time. The factory pattern, and several related patterns, provide a solution and allow us, in effect, have virtual constructors.

Chapter 13, *Memory Ownership*, Memory mismanagement is one of the more common problems in C++ programs. Many of these problems boil down to incorrect assumptions about which part of code or which entity owns a particular memory. Then we get memory leaks, accessing unallocated memory, excessive memory use, and other difficult to debug problems. Modern C++ has a set of memory ownership idioms, which, taken together, allow the programmer to clearly express the design intent when it comes to memory ownership. This, in turn, makes it much easier to write code that correctly allocates, accesses, and deallocates memory.

Chapter 14, *Template Method Pattern and Non-Virtual Idiom*, Template is one of the classic patterns, which, in C++, has a sort of "local flavor" known as the Non-Virtual Interface idiom (NVI). This chapter explains the use of the template pattern in C++, and the the outcome that appears rather non-intuitive at first: private virtual functions.

Chapter 15, *Curiously Recurring Template Pattern*, This chapter demonstrates one of the more convoluted C++ patterns, the Curiously Recurring Template Pattern (CRTP). On one hand, CRTP is a kind of Delegation pattern, where certain aspect of behavior is delegates to derived classes. On the other hand, it is often used to implement static (compile-time) polymorphism without the overhead of dynamic polymorphism. We will continue exploration of the CRTP in the next chapter, Mixins.

Chapter 16, *Mixin*, A mixin is a "class fragment" that is intended to add a certain functionality or property. This chapter explains how mixin design is implemented in C++, and serves as a foundation for the latter chapter on policy-based design.

Chapter 17, *Scopeguard*, The scopeguard pattern is a more generic version of the RAII idiom, and it addresses some of the annoyances and limitations we have encountered while exploring that idiom. This chapter will show why a "generic RAII" is desirable, then show possible implementations in different versions of C++. This pattern uses multiple idioms from the previous part: RAII, template argument deduction, type erasure, exception-safe swap.

Chapter 18, *Serialization/Deserialization and Virtual Template*, Static (templates) and dynamic (virtual functions) polymorphism in C++ just don't mix. Template member functions cannot be virtual. But sometimes you really want them to be. Serialization/deserialization of objects is one application: it is more reliable to have both serialization and deserialization functions be generated from the same template, it makes the code more robust. But then how can we serialize polymorphic types? There are several related idioms that help, and we cover them in this chapter.

Chapter 19, *Policy-Based Design*, This chapter explores one of the most complex and powerful C++ patterns. Policy-based design allows nearly unlimited customization of library classes by the users of the library. It is a way to provide a wide range 9f behaviors with very few lines of code. It can be viewed as a pinnacle of C++ design, in that it combines so many other patterns and idioms: strategy, mixin, CRTP, adapters, type erasure, and more. But it comes at a price.

Chapter 20, *Policy-Based Design, Inside-Out*, If the previous

chapter on policy-based design left you stunned or terrified, this chapter may provide some relief. There is an alternative to the policy-based design, and it comes from using the classic decorator pattern, again in the C++ style: it is applied to static polymorphism instead of dynamic one. This is a less complex, and less powerful, design pattern, but it often can solve the same type of problems.

Chapter 21, *Typelist and Typemap*, Typelist, or a container of types - a compile-time entity - is a C++-specific pattern. It was first proposed, and implemented, when we were discovering the hidden possibilities of the template metaprogramming in C++, way before C++11. It worked, too, in a very LISP sort of way. The most useful application of it was probably an entry for a challenge to write the most difficult to understand C++ program. C++11 gave us a variadic template, which made typelists at least look like they belong to C++. This chapter introduces the typelist pattern and explains its possible applications. We then generalize typelist into a typemap container. This pattern offers a solution to one of the problems of the policy-based design: he verbose way in which a policy has to be specified. Now that we have explored the typelist pattern, we can apply it to simplify the use of policy-based classes.

Chapter 22, *Visitor and Multiple Dispatch*, This chapter returns to one of the classic patterns, the Visitor. Visitor, in object-oriented programming, implements the double dispatch: the runtime behavior is determined by two dynamic types (as opposed to a virtual function call that implements single dispatch which depends only on the type of the calling object). The generalization of this pattern is multiple dispatch, where the an action is performed on a set of polymorphic objects and depends on the types of all of these objects. C++ does not have double dispatch natively in the language but it can be implemented fairly easily. Multiple dispatch is possible, but gets

progressively more verbose. Of course, in C++ it is also possible to apply multiple dispatch to the static polymorphism, and this problem is handled with the full power of C++ templates.

An Introduction to Inheritance and Polymorphism

Coming soon...

Function Templates

Coming Soon...

Class Templates

Coming Soon...

Swap

Coming soon...

A Comprehensive Look at RAII

Resource management is probably the second most frequent thing a program does, after computing. Note that just because it's frequent, does not mean it's visible: some languages hide much, or all, of the resource management from the user. Just because it is hidden, does not mean it's not there. Every program needs to use some memory, and memory is a resource. A program would be of no use if it never interacted with the outside world in some way, at least to print the result, and input and output channels (files, sockets) are resources. C++, with its zero-overhead abstraction philosophy, does not hide the resources or their management at the core language level. But don't confuse hiding resources with managing them.

The following topics will be covered in this chapter:

- What is considered a "resource" in a C++ program?
- What are the key concerns for managing resources in C++?
- What is the standard approach to managing resources in C++ (the RAII)?
- How does RAII solve the problems of resource management?
- What precautions must be taken when writing RAII objects?

- What are the consequences of using RAII for resource management?

Technical requirements

GoogleTest unit testing framework: `https://github.com/google/googletest`

Google Benchmark library: `https://github.com/google/benchmark`

Example code: `https://github.com/PacktPublishing/Hands-On-Design-Patterns-with-C-/tree/master/Ch05`

Resource management in C++

Every program operates on resources and needs to manage them. The most commonly used resource is the memory, of course. Hence you often read about "memory management" in C++. But really, resources can be just about anything. Many programs exist specifically to manage real, tangible physical resources, or the more ephemeral but no less valuable digital ones. Money in bank accounts, airline seats, car parts and assembled cars, or crates of milk, in today's world, if it is something that needs to be counted and tracked, there is a piece of software somewhere that is doing it. But even in a program that does pure computations, there may be varied and complex resources, unless the program also eschews abstractions and operates at the level of bare numbers. For example, a physics simulation program may have particles as resources.

All of these resources have one thing in common: they need to be accounted for. They should not vanish without a trace, and a program should not just make up resources that don't really exist. Often, a specific instance of a resource is needed: you would not want someone else's purchase to be debited from your bank account, the specific instance of the resource matters. Thus, the most important consideration when evaluating different approaches to resource management is correctness: how well does the design ensure that resources are managed properly, how easy is it to make a mistake, and how hard would it be to find it? It should come as no surprise, then, when we use a testing framework to present the coding examples of resource management in this chapter.

Installing GoogleTest

We will be testing for correctness very small fragments of code. On the one hand, this is simply because each fragment illustrates a specific concept or idea. On the other hand, even in a large-scale software system, resource management is done by small building blocks of code. They may combine to form a quite complex resource manager, but each block performs a specific function and is testable. The appropriate testing system for his situation is a unit testing framework. There are many such to choose from; in this book, we will use the GoogleTest unit testing framework. To follow along with the examples in this chapter, you must first download and install the framework (follow the instructions in the Readme file). Then you can compile and run the examples. You can build the sample tests included with the library to see how to build and link with GoogleTest on your particular system. For example, on a Linux machine, the command to build and run a test `memory1.C` might look something like this:

```
$CXX memory1.C -I. -I$GTEST_DIR/include -g -O0 -I. -Wall -Wextra -
Werror -pedantic --std=c++14 $GTEST_DIR/lib/libgtest.a
$GTEST_DIR/lib/libgtest_main.a -lpthread -lrt -lm -o memory1 &&
./memory1
```

Here `$CXX` is your C++ compiler, such as `g++` or `g++-6`, and `$GTEST_DIR` is the directory where GoogleTest is installed.

Counting resources

A unit testing framework, such as GoogleTest, allows us to execute some code and verify that the results are what they should be. The "results" we can look at including any variable or expression we can access from the test program. That definition does not extend to, for example, the amount of memory that is currently in use. So, if we want to verify that resources are not disappearing, we have to count them.

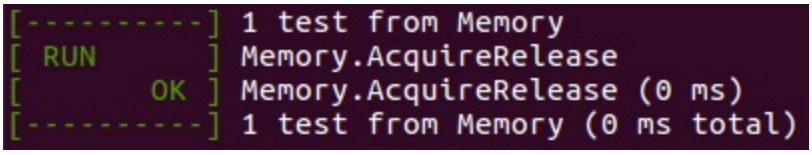
In the simple test fixture below, we use a special resource class instead of, say, an `int`. This class is instrumented to count how many objects of this type have been created, and how many are currently alive:

```
struct object_counter {
    static int count;
    static int all_count;
    object_counter() { ++count; ++all_count; }
    ~object_counter() { --count; }
};
```

Now we can test that our program manages resources correctly:

```
TEST(Scoped_ptr, Construct) {
    object_counter::all_count = object_counter::count = 0;
    object_counter* p = new object_counter;
    EXPECT_EQ(1, object_counter::count);
    EXPECT_EQ(1, object_counter::all_count);
    delete p;
    EXPECT_EQ(0, object_counter::count);
    EXPECT_EQ(1, object_counter::all_count);
}
```

In GoogleTest, every test is implemented as a "test fixture." There are several types; the simplest one is a standalone test function like the one we use here. The expected results are verified using one of the `EXPECT_*` macros and any test failures will be reported. This test verifies that, after creating and deleting an instance of the type `object_counter`, there are no such objects left, and that exactly one was constructed. Running this simple test program tells us that the test has passed:



Dangers of manual resource management

C++ allows us to manage resources almost at the hardware level, and, someone, somewhere, must indeed manage them at this level. The latter is actually true for every language, even the high-level ones that do not expose such details to the programmers. But the "somewhere" does not have to be in your program! Before we learn the C++ solutions and tools for resource management, we should first understand the problems that arise from not using any such tools.

Manual resource management is error-prone

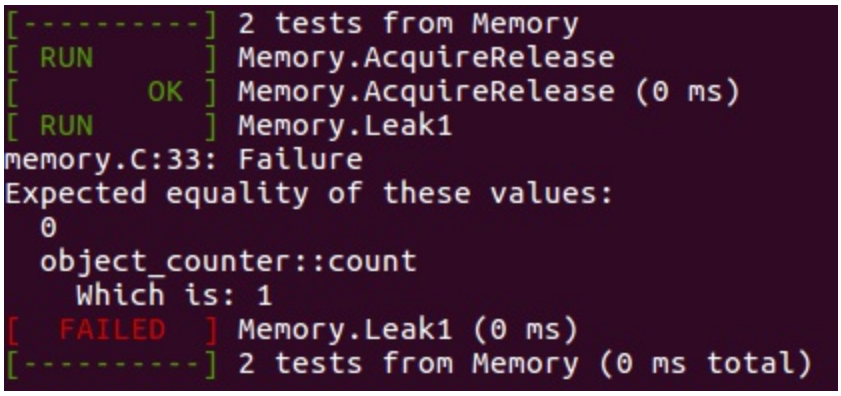
The first, obvious, danger of managing every resource manually, with explicit calls to acquire and release each one, is that it is easy to forget the latter:

```
{
    object_counter* p = new object_counter;
      ... many more lines of code ...
 } // Were we supposed to do something here? Can't remember now...
```

We are now leaking a resource (`object_counter` objects, in this case). If we did this in a unit test, it would fail:

```
TEST(Memory, Leak1) {
    object_counter::all_count = object_counter::count = 0;
    object_counter* p = new object_counter;
    EXPECT_EQ(1, object_counter::count);
    EXPECT_EQ(1, object_counter::all_count);
    //delete p;    // Forgot that
    EXPECT_EQ(0, object_counter::count);    // This test fails now!
    EXPECT_EQ(1, object_counter::all_count);
}
```

You can see the failing tests, and the location of the failures, reported by the unit test framework:



In a real program, finding such errors is much harder. Memory debuggers and sanitizers can help with memory leaks, but they require that the program actually execute the buggy code, so they depend on the test coverage.

The resource leaks can be much more subtle and harder to find, too. Consider this code, where we did not forget to release the resource:

```
bool process(... some parameters ... ) {
    object_counter* p = new object_counter;
      ... many more lines of code ...
    delete p;        // A-ha, we remembered!
    return true;     // Success
 }
```

During the subsequent maintenance, a possible failure condition was discovered, and the appropriate test added:

```
bool process(... some parameters ... ) {
    object_counter* p = new object_counter;
      ... many more lines of code ...
    if (!success) return false;    // Failure, cannot continue
      ... even more lines of code ...
    delete p;        // Still here
    return true;     // Success
 }
```

This change introduced a subtle bug: now resources are leaked only if the intermediate computation has failed and triggered the early return. If the failure is rare enough, this mistake may escape all tests, even if the testing process employs regular memory sanitizer runs. It is also all too easy to make since the edit could be made in a place far removed from both the construction and deletion of the object, and nothing in the immediate context gives the programmer a hint that a resource needs to be released.

The alternative to leaking resource, in this case, is to release it. Note that this leads to some code duplication:

```
bool process(... some parameters ... ) {
    object_counter* p = new object_counter;
      ... many more lines of code ...
    if (!success) { delete p; return false; }   // Failure, cannot
continue
      ... even more lines of code ...
    delete p;        // Still here
    return true;     // Success
 }
```

As with any code duplication, comes the danger of code divergence. Say, the next round of code enhancements required more than one `object_counter`, and an array of them is now allocated:

```
bool process(... some parameters ... ) {
    object_counter* p = new object_counter[10];    // Now an array
      ... many more lines of code ...
    if (!success) { delete p; return false; }   // Old scalar
delete
      ... even more lines of code ...
    delete [] p;     // Matching array delete
    return true;     // Success
 }
```

If we change `new` to the array `new`, we must change the `delete` as well; the thought goes, there is probably one at the end of the function. Who knew that there is one more in the middle? Even if the programmer had not forgotten about the resources, the manual resource management gets disproportionately more error-prone as the program becomes more complex. And not all resources are as forgiving as some counter object. Consider this code that performs some concurrent computation and must acquire and release mutex locks. Note the very words "acquire" and "release," the common terminology for locks, suggest that locks are treated as a kind of resource (the resource here is exclusive access to the data protected by the lock).

```
std::mutex m1, m2, m3;
bool process_concurrently(... some parameters ... ) {
    m1.lock();
    m2.lock();
    ... need both locks in this section ...
    if (!success) {
        m1.unlock();
        m2.unlock();
        return false;
    } // Both locks unlocked
    ... more code ...
    m2.unlock();        // Don't need exclusive access guarded by
this mutex, still need m1
    m3.lock();
    if (!success) {
        m1.unlock();
        return false;
    } // No need to unlock m2 here
    ... more code ...
    m1.unlock(); m3.unlock();
    return true;
 }
```

This code has both the duplication and the divergence. It also has a bug, see if you can find it (hint - count how many times `m3` is unlocked vs how many `return` statements there are after it's

locked). As the resources become more numerous and complex to manage, such bugs are going to creep up more often.

Resource management and exception safety

Remember the code at the beginning of the previous section, the one we said is correct, the one where we did not forget to release the resource?

```
bool process(... some parameters ... ) {
    object_counter* p = new object_counter;
      ... many more lines of code ...
    delete p;
    return true;     // Success
 }
```

I have bad news for you: this code probably wasn't correct either. If any of the many more lines of code can throw an exception, then `delete p` is never going to be executed:

```
bool process(... some parameters ... ) {
    object_counter* p = new object_counter;
      ... many more lines of code ...
    if (!success) throw process_exception();    // Exceptional
circumstances, cannot continue
      ... even more lines of code ...
    delete p;        // Still here but won't do anything if
exception is thrown
    return true;
 }
```

This looks very similar to the early `return` problem, only worse: the exception can be thrown by any code that the `process()` function calls. The exception can even be added later to some

code that the `process()` function calls, without any changes in the function itself. It used to work fine, then one day it does not.

Unless we change our approach to resource management, the only solution is to use try-catch blocks:

```
bool process(... some parameters ... ) {
    object_counter* p = new object_counter;
    try {
          ... many more lines of code ...
        if (!success) throw process_exception();    // Exceptional
circumstances, cannot continue
          ... even more lines of code ...
    } catch ( ... ) {
        delete p;    // For exceptionan case
    }
    delete p;        // For normal case
    return true;
 }
```

The obvious problem here is code duplication, again, as well as the proliferation of the try-catch blocks literally everywhere. Worse than that, this approach does not scale if we need to manage multiple resources, or just manage anything more complex than a single acquisition with a corresponding release:

```
std::mutex m;
bool process(... some parameters ... ) {
    m.lock();
    object_counter* p = new object_counter;    // Problem #1:
constructor can throw
    try {
          ... many more lines of code ...
        m.unlock();                            // Critical section
ends here
          ... even more lines of code ...
    } catch ( ... ) {
        delete p;      // OK, always needed
        m.unlock();    // Do we need this? Depends on where the
exception was thrown!
```

```
        throw;          // Rethrow the exception for the client to
handle
    }
    delete p;        // For normal case, no need to unlock mutex
    return true;
 }
```

Now we can't even decide whether the catch block should release the mutex or not: it depends on whether the exception was thrown before or after the `unlock()` that happens in the normal, non-exceptional control flow. Also, `object_counter` constructor could throw an exception (not the simple one we had so far, but a more complex one that ours could evolve into). That would happen outside of the try-catch block, and the mutex would never get unlocked.

It should be clear to the reader by now that we need an entirely different solution for the resource management problem, not some patchwork. The next section introduces the pattern that became "golden standard" of resource management in C++.

Resource Acquisition is Initialization (RAII) idiom

We have seen in the previous section how the ad-hock attempts to manage resources become unreliable, error-prone, and eventually fail. What we need is to make sure that resource acquisition is always paired up with resource release, and that these two actions happen before and after the section of code that uses the resource, respectively. In C++ this kind of bracketing of a code sequence by a pair of actions is known as the Execute Around design pattern (see the article "C++ Patterns: Executing Around Sequences" by Kevlin Henney, `http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/ExecutingAroundSequences.pdf`). When specifically applied to resource management, this pattern is much wider known as Resource Acquisition is the Initialization idiom (RAII).

RAII in a nutshell

The basic idea behind the RAII is very simple: there is one kind of function in C++ that is guaranteed to be called automatically, and that is the destructor of an object created on the stack, or the destructor of an object that is a data member of another object (in the latter case, the guarantee holds only if the containing class itself is destroyed). If we could hook up the release of the resource to the destructor of such object, then the release could not be forgotten or skipped. It stands to reason that if releasing the resource is handled in the destructor, acquiring it should be handled by the constructor, i.e., during the initialization of the object. Hence the name, Resource Acquisition is Initialization. Let us see how this works in the simplest case of memory allocation (via `operator new`). First, we need a class that can be initialized from a pointer to the newly allocated object, and whose destructor will delete that object:

```
template <typename T>
class raii {
    public:
    explicit raii(T* p) : p_(p) {}
    ~raii() { delete p_; }
    private:
    T* p_;
};
```

Now it is very easy to make sure that deletion is never omitted, and we can verify that it works as expected with a test that uses our `object_counter`:

```
TEST(RAII, AcquireRelease) {
```

```
    object_counter::all_count = object_counter::count = 0;
    {
        raii<object_counter> p(new object_counter);
        EXPECT_EQ(1, object_counter::count);
        EXPECT_EQ(1, object_counter::all_count);
    } // No need to delete p, it's automatic
    EXPECT_EQ(0, object_counter::count);
    EXPECT_EQ(1, object_counter::all_count);
}
```

Note that in C++17 the class template type is deduced from the constructor and we can write simply

```
raii p(new object_counter);
```

Of course, we probably want to use the new object for more than creating and deleting it, so it would be nice to have access to the pointer stored inside the RAII object. There is no reason to grant such access in any way other than the standard pointer syntax, which makes our RAII object a kind of pointer itself:

```
template <typename T>
class scoped_ptr {
    public:
    explicit scoped_ptr(T* p) : p_(p) {}
    ~scoped_ptr() { delete p_; }
    T* operator->() { return p_; }
    const T* operator->() const { return p_; }
    T& operator*() { return *p_; }
    const T& operator*() const { return *p_; }
    private:
    T* p_;
};
```

This pointer can be used to automatically delete, at the end of the scope, the object it points to (hence the name):

```
TEST(Scoped_ptr, AcquireRelease) {
```

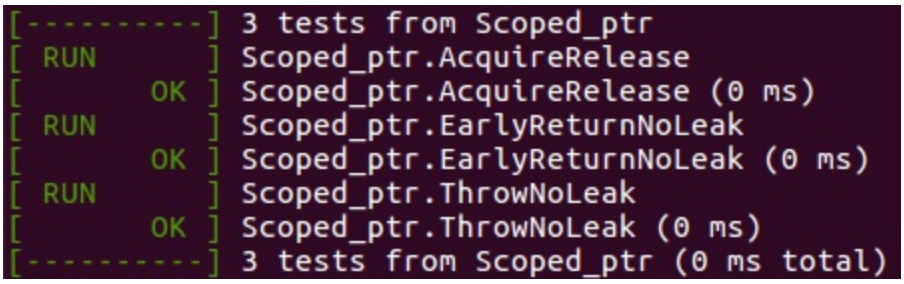
```
    object_counter::all_count = object_counter::count = 0;
    {
        scoped_ptr<object_counter> p(new object_counter);
        EXPECT_EQ(1, object_counter::count);
        EXPECT_EQ(1, object_counter::all_count);
    }
    EXPECT_EQ(0, object_counter::count);
    EXPECT_EQ(1, object_counter::all_count);
}
```

The destructor is called when the scope containing the `scoped_ptr` object is exited. It does not matter how it is exited: an early `return` from a function, a `break` or `continue` statement in the loop, or an exception being thrown are all handled in exactly the same way, and without leaks. We can verify this with tests, of course:

```
TEST(Scoped_ptr, EarlyReturnNoLeak) {
    object_counter::all_count = object_counter::count = 0;
    do {
        scoped_ptr<object_counter> p(new object_counter);
        break;
    } while (false);
    EXPECT_EQ(0, object_counter::count);
    EXPECT_EQ(1, object_counter::all_count);
}

TEST(Scoped_ptr, ThrowNoLeak) {
    object_counter::all_count = object_counter::count = 0;
    try {
        scoped_ptr<object_counter> p(new object_counter);
        throw 1;
    } catch ( ... ) {
    }
    EXPECT_EQ(0, object_counter::count);
    EXPECT_EQ(1, object_counter::all_count);
}
```

All tests pass, confirming that there is no leak:



Similarly, we can use a scoped pointer as a data member in another class, a class that has secondary storage and must release it upon destruction:

```
class A {
    public:
    A(object_counter* p) : p_(p) {}
    private:
    scoped_ptr<object_counter> p_;
};
```

This way, we don't have to delete the object manually in the destructor of class `A`, and, in fact, if every data member of class `A` takes care of itself in a similar fashion, class `A` may not even need an explicit destructor.

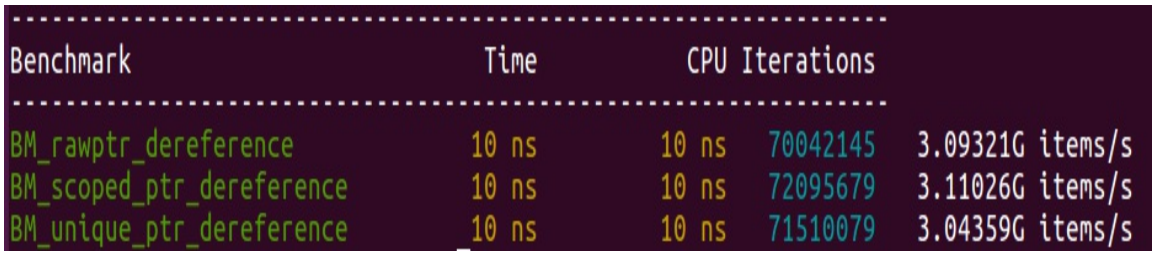
Anyone familiar with C++11 should recognize our `scoped_ptr` as a very rudimentary version of `std::unique_ptr`, which can be used for the same purpose. As one might expect, the standard unique pointer's implementation has a lot more to it, and for good reasons. We will review some of these reasons later in this chapter.

One last issue to consider is that of performance. C++ strives for zero-overhead abstractions, whenever possible. In this case, we are wrapping a raw pointer into a smart pointer object. However, the compiler does not need to generate any additional machine instructions; the wrapper only forces the compiler to generate the code that, in a correct program, it would have done anyway.

We can confirm with a simple benchmark that both the construction/deletion and the dereference of our `scoped_ptr` (or `std::unique_ptr`, for that matter) take exactly the same time as the corresponding operations on a raw pointer. For example, the following micro-benchmark (using Google benchmark library) compares the performance of all three pointer types for dereferencing:

```
void BM_rawptr_dereference(benchmark::State& state) {
    int* p = new int;
    for (auto _ : state) {
        REPEAT(benchmark::DoNotOptimize(*p);)
    }
    delete p;
    state.SetItemsProcessed(32*state.iterations());
}
void BM_scoped_ptr_dereference(benchmark::State& state) {
    scoped_ptr<int> p(new int);
    for (auto _ : state) {
        REPEAT(benchmark::DoNotOptimize(*p);)
    }
    state.SetItemsProcessed(32*state.iterations());
}
void BM_unique_ptr_dereference(benchmark::State& state) {
    std::unique_ptr<int> p(new int);
    for (auto _ : state) {
        REPEAT(benchmark::DoNotOptimize(*p);)
    }
    state.SetItemsProcessed(32*state.iterations());
}
BENCHMARK(BM_rawptr_dereference);
BENCHMARK(BM_scoped_ptr_dereference);
BENCHMARK(BM_unique_ptr_dereference);
BENCHMARK_MAIN();
```

The benchmark shows that the smart pointers indeed incur no overhead:



```
-------------------------------------------------------------
Benchmark                   Time           CPU Iterations
-------------------------------------------------------------
BM_rawptr_dereference      10 ns         10 ns   70042145   3.09321G items/s
BM_scoped_ptr_dereference  10 ns         10 ns   72095679   3.11026G items/s
BM_unique_ptr_dereference  10 ns         10 ns   71510079   3.04359G items/s
```

We have covered in enough details the application of RAII to managing memory. But there are other resources a C++ program needs to manage and keep track of, so we have to expand our view of the RAII now.

RAII for other resources

The name, RAII, refers to "resources" and not "memory," and indeed the same approach is applicable to other resources. For each resource type, we need a special object, although generic programming and lambda expressions may help us to write less code (more on this in the latter chapter on ScopeGuard pattern). The resource is acquired in the constructor and released in the destructor. Note that there are two slightly different flavors of the RAII. The first option is the one we have already seen: the actual acquisition of the resource is done at initialization but outside of the constructor of the RAII object. The constructor merely captures a handle (such as a pointer) that resulted from this acquisition. This was the case in the `scoped_ptr` we just saw: memory allocation and object construction were both done outside of the constructor of the `scoped_ptr` object, but still during its initialization. The second option is for the constructor of the RAII object to actually acquire the resource. Let us see how this works on the example of an RAII object that manages mutex locks:

```
class mutex_guard {
    public:
    explicit mutex_guard(std::mutex& m) : m_(m) { m_.lock(); }
    ~mutex_guard() { m_.unlock(); }
    private:
    std::mutex& m_;
};
```

Here the constructor of the `mutex_guard` class itself acquires the resource, in this case, exclusive access to the shared data protected by the mutex. The destructor releases that resource.

Again, this pattern completely removes the possibility of "leaking" a lock (exiting a scope without releasing the lock), for example, when an exception is thrown:

```
std::mutex m;
TEST(Scoped_ptr, ThrowNoLeak) {
    try {
        mutex_guard lg(m);
        EXPECT_FALSE(m.try_lock());           // Expect to be
locked already
        throw 1;
    } catch ( ... ) {
    }
    EXPECT_TRUE(m.try_lock()); m.unlock();    // Expect to be
unlocked, try_lock() will lock
}
```

In this test, we check whether the mutex is locked or not by calling `std::mutex::try_lock()`: we cannot call `lock()` if the mutex is already locked, it will deadlock. By calling `try_lock()`, we can check the state of the mutex without the risk of deadlock (but remember to unlock the mutex if `try_lock()` succeeds).

Again, the standard provides an RAII object for mutex locking, `std::lock_guard`. It is used in a similar manner but can be applied to any mutex type that has `lock()` and `unlock()` member functions.

Releasing early

The scope of a function or a loop body does not always match the desired duration of holding the resource. If we do not want to acquire the resource at the very beginning of the scope, this is easy: the RAII object can be created anywhere, not just at the beginning of the scope. Resources are not acquired until the RAII object is constructed:

```
void process(...) {
    ... do work that does not need exclusive access ...
    mutex_guard lg(m);            // Now we lock
    ... work on shared data, now protected by mutex ...
} // lock is released here
```

However, the release still happens at the end of the function body scope. What if we only want to lock a short portion of code inside the function? The simplest answer is to create an additional scope:

```
void process(...) {
    ... do work that does not need exclusive access ...
    {
        mutex_guard lg(m);            // Now we lock
        ... work on shared data, now protected by mutex ...
    } // lock is released here
    ... more non-exclusive work ...
}
```

It may be surprising if you have never seen it before, but in C++ any sequence of statements can be enclosed in the curly braces `{ ... }`. Doing so creates a new scope with its own local variables.

Unlike the curly braces that come after loops or conditional statements, the only purpose of this scope is controlling the lifetime of these local variables. A program that uses RAII extensively often has many such scopes, enclosing variables with different lifetimes that are shorter than the overall function or loop body. This practice also improves readability by making it clear that some variables will not be used after a certain point, the reader does not need to scan the rest of the code looking for possible references to these variables. Also, one cannot accidentally add such reference by mistake, if the intent was to "expire" a variable and never use it again.

But what if a resource may be released early, but only if certain conditions are met? One possibility is, again, to contain the use of the resource in a scope, and exit that scope when the resource is no longer needed. It would be convenient to be able to `break` out of a scope. A common way to do just that is to write a do-once loop:

```
void process(...) {
    ... do work that does not need exclusive access ...
    do {                              // Not really a loop
        mutex_guard lg(m);            // Now we lock
        ... work on shared data, now protected by mutex ...
        if (work_done) break;         // Exit the scope
        ... work on the shared data some more ...
    } while (false);                  // lock is released here
    ... more non-exclusive work ...
}
```

However, this approach does not always work (we may want to release the resources but not other local variables we defined in the same scope), and the readability of the code suffers as the control flow gets more complex. Resist the impulse to accomplish this by allocating the RAII object dynamically, with `operator new`! This completely defeats the whole point of RAII, since you now must remember to invoke `operator delete`. We can

enhance our resource-managing objects by adding a client-triggered release, in addition to the automatic release by the destructor. We just have to make sure that the same resource is not released twice. For example, for the `scoped_ptr`:

```
template <typename T>
class scoped_ptr {
    public:
    explicit scoped_ptr(T* p) : p_(p) {}
    ~scoped_ptr() { delete p_; }
      ...
    void reset() { delete p_; p_ = nullptr; }    // Releases
resource early
    private:
    T* p_;
};
```

After calling `reset()`, the object managed by the `scoped_ptr` is deleted, and the pointer data member of the `scoped_ptr` object is reset to null. Note that we did not need to add a condition check to the destructor because calling delete on a null pointer is allowed by the standard, it does nothing. The resource is released only once, either explicitly by the `reset()` call, or implicitly at the end of the scope containing the `scoped_ptr` object.

For the `mutex_guard` class, we can't deduce from just the lock whether an early release was called or not, and we need an additional data member to keep track of that:

```
class mutex_guard {
    public:
    explicit mutex_guard(std::mutex& m) : m_(m), must_unlock_(true)
{ m_.lock(); }
    ~mutex_guard() { if (must_unlock_) m_.unlock(); }
    void reset() { m_.unlock(); must_unlock_ = false; }
    private:
    std::mutex& m_;
    bool must_unlock_;
```

```
};
```

Now we can verify that the mutex is released only once, at the right time, with this test:

```
TEST(mutex_guard, Reset) {
    {
        mutex_guard lg(m);
        EXPECT_FALSE(m.try_lock());
        lg.reset();
        EXPECT_TRUE(m.try_lock()); m.unlock();
    }
    EXPECT_TRUE(m.try_lock()); m.unlock();
}
```

The standard `std::unique_ptr` supports `reset()`, the `std::lock_guard` does not, so if you need to release a mutex early, you need to write your own guard object. Fortunately, a lock guard is a pretty simple class, but finish reading this chapter before you start writing, there are a few gotchas to keep in mind.

Note that the `reset()` method of `std::unique_ptr` actually does more than just delete the object prematurely. It can also be used to "reset" the pointer: make it point to a new object, while the old one is deleted. It works something like this (the actual implementation in the standard is a bit more complex because of the additional functionality the unique pointer has):

```
template <typename T>
class scoped_ptr {
    public:
    explicit scoped_ptr(T* p) : p_(p) {}
    ~scoped_ptr() { delete p_; }
      ...
    void reset(T* p = nullptr) { delete p_; p_ = p; }    // Reseat
the pointer
    private:
    T* p_;
```

```
};
```

Note that this code breaks if a scoped pointer is reseated to itself (i.e., if `reset()` is called with the same value as what is stored in `p_`). We could check for this condition and do nothing; it is worth noting that the standard does not require such check for `std::unique_ptr`.

Careful implementation of RAII objects

It is, obviously, very important that the resource management objects do not mismanage resources they are entrusted to guard. Unfortunately, the simple RAII objects we have been writing so far have several glaring holes.

The first problem arises when someone tries to copy these objects. Each of the RAII objects we have considered in this chapter is responsible for managing a unique instance of its resource, and yet, nothing prevents us from copying this object:

```
scoped_ptr<object_counter> p(new object_counter);
scoped_ptr<object_counter> p1(p);
```

This code invokes the default copy constructor, which simply copies the bits inside the object, in our case the pointer to the `object_counter`. Now we have two RAII objects that both control the same resource. Two destructors will be called, eventually, and both will attempt to delete the same object. The second deletion is undefined behavior (if we are very fortunate, the program will crash at that point).

Assignment of RAII objects is similarly problematic:

```
scoped_ptr<object_counter> p(new object_counter);
scoped_ptr<object_counter> p1(new object_counter);
p = p1;
```

The default assignment operator also copies the bits of the object. Again, we have two RAII objects that will delete the same managed object. Equally troublesome is the fact that we have no RAII objects that manage the second `object_counter`, the old pointer inside `p1` is gone, and there is no other reference to this object, so no way to delete it.

The `mutex_guard` does no better: an attempt to copy it results in two mutex guards that will unlock the same mutex. The second unlock will be done on a mutex that is not locked (at least not by the calling thread), which, according to the standard, is undefined behavior. Assignment of the mutex_guard object is not possible, though, because the default assignment operator is not generated for objects with reference data members.

As the reader has probably noticed, the problem is created by the **default** copy constructor and **default** assignment operator. Does it mean that we should have implemented our own? What would they do? Only one destructor should be called for each object that was constructed; a mutex can only be unlocked once after it was locked. This suggests that an RAII object should not be copied at all, and we should disallow both copying and assignment:

```
template <typename T>
class scoped_ptr {
    public:
    explicit scoped_ptr(T* p) : p_(p) {}
    ~scoped_ptr() { delete p_; }
      ...
    private:
    T* p_;
    scoped_ptr(const scoped_ptr&) = delete;
    scoped_ptr& operator=(const scoped_ptr&) = delete;
};
```

Deleting default member functions is a C++11 feature; before

that, we would have to declare, but not define, both functions as private:

```
template <typename T>
class scoped_ptr {
      ...
    private:
    scoped_ptr(const scoped_ptr&);             // No {} - not
defined!
    scoped_ptr& operator=(const scoped_ptr&);
};
```

There are some RAII objects that can be copied. These are reference-counted resource management objects; they keep track of how many copies of the RAII object exist for the same instance of the managed resource. The last of the RAII objects has to release the resource when it is deleted. We discuss shared management of resources in more details in the later chapter on memory ownership.

A different set of considerations exist for move constructor and assignment. Moving the object does not violate the assumption that there is only one RAII object that owns a particular resource. It merely changes which RAII object that is. In many cases, such as mutex guards, it does not make sense to move an RAII object (indeed, the standard does not make `std::lock_guard` movable). Moving a unique pointer is possible and makes sense in some contexts, which we also explore in the chapter on memory ownership. However, for a scoped pointer, moving would be undesirable as it allows to extend the lifetime of the managed object beyond the scope where it was created. Note that we do not need to delete move constructor or move assignment operator if we already deleted the copying ones (although doing so does no harm). On the other hand, `std::unique_ptr` is a movable object, which means using it as a scope-guarding smart pointer does not offer the same protection, the resource could be moved

out. However, if you need a scoped pointer, there is a very simple way to make std::unique_ptr do this job perfectly: all you have to do is to declare a `const std::unique_ptr` object:

```
std::unique_ptr<int> p;
{
    std::unique_ptr<int> q(new int);          // Can be moved out
of the scope
    q = std::move(p);                         // and here it
happens
    const std::unique_ptr<int> r(new int);    // True scoped
pointer, not going anywhere
    q = std::move(r);                         // Does not compile
}
```

So far, we have protected our RAII objects against duplicating or losing resources. But there is one more king of resource management mistake that we have not yet considered. It seems obvious that a resource should be released in a way that matches its acquisition. And yet, nothing protects our `scoped_ptr` from such a mismatch between construction and deletion:

```
scoped_ptr<int> p(new int[10]);
```

The problem here is that we have allocated multiple objects using array version of the `operator new`; it should be deleted with the array version of `operator delete`: `delete [] p_` must be called inside the `scoped_ptr` destructor, instead of `delete p_` that we have there now.

More generally, an RAII object that accepts a resource handle during initialization instead of acquiring the resource itself (like the `mutex_guard` does) must somehow ensure that the resource is released in the "right" way that matches the way it was acquired. Obviously, this is not possible, in general. In fact, it is impossible to do automatically even for the simple case of mismatched array

`new` and scalar `delete` (`std::unique_ptr` does no better than our `scoped_ptr` here, although facilities like `std::make_unique` make writing such code less error-prone). In general, either the RAII class is designed to release resources in one particular way, or the caller must specify how the resource must be released. The former is certainly easier, and in many cases is quite sufficient. In particular, if the RAII class also acquires the resource, like our `mutex_guard`, it certainly knows how to release it. Even for the scoped_ptr, it would not be too hard to create two versions, `scoped_ptr` and `scoped_array`; the second one is for objects allocated by the array `operator new`. A more general version of an RAII class is parametrized not just by the resource type, but also by a callable object used to release this type, usually known as the deleter. The deleter can be a function pointer, a member function pointer, or an object with `operator()` defined - basically, anything that can be called like a function. Note that the deleter has to be passed to the RAII object in its constructor, and stored inside the RAII object, which makes the object larger. Also, the type of the deleter is a template parameter of the RAII class, unless it is erased from the RAII type (we will have a whole chapter later dedicated to the type erasure).

Downsides of RAII

Honestly, there aren't any significant downsides to RAII. It is by far the most widely used idiom for resource management in C++. The only issue of significance to be aware of has to do with exceptions. Releasing a resource can fail, like anything else. The usual way in C++ to signal a failure is to throw an exception. When that is undesirable, we fall back on returning error codes from functions. With RAII, we can do neither of these things.

It is easy to understand why error codes are not an option: the destructor does not return anything. Also, we cannot write the error code into some status data member of the object since the object is being destroyed, its data members are gone. So are the other local variables from the scope containing the RAII object. The only way to save an error code for the future examination would be to write it into some sort of a global status variable, or at least a variable from the containing scope. Possible in a bind, but very inelegant and error-prone. This is exactly the problem C++ was trying to solve when exceptions were introduced: manually propagated error codes are error-prone and unreliable.

So, if the exceptions are the answer to error reporting in C++, why not use them here? The usual answer is "because the destructors in C++ cannot throw." This captures the gist of it, but the real restriction is a bit more nuanced. First of all, prior to C++11, the destructors technically could throw, the exception would propagate and (hopefully) eventually get caught and processed. In C++11 all destructors are, by default, `noexcept`, unless explicitly specified as `noexcept(false)`. If a `noexcept` function throws, the program immediately terminates. So indeed, in C++11,

destructors cannot throw unless you specifically allow them to do it. But what's wrong with throwing an exception in the destructor? If the destructor is executed because the object was deleted, or because the control reached the end of the scope for a stack object, nothing. The "wrong" happens if the control did not reach the end of the scope normally and the destructor is executed because an exception was already thrown. In C++, two exceptions cannot propagate at the same time. If this happens, the program will immediately terminate (note that a destructor can throw and catch an exception, there is no problem with that as long as that exception does not propagate out of the destructor). Of course, when writing a program, there is no way to know when some function called from something in a particular scope, could throw. If the resource release throws and the RAII object allows that exception to propagate out of its destructor, the program is going to terminate if that destructor was called during exception handling. The only safe way is never to allow exceptions to propagate from a destructor. This does not mean that the function that releases the resource itself cannot throw, but, if it does, an RAII destructor has to catch that exception:

```
class raii {
    ...
    ~raii() {
        try {
            release_resource();     // Might throw
        } catch ( ... ) {
            ... handle the exception, do NOT rethrow ...
        }
    }
};
```

This still leaves us with no way to signal that an error happened during resource release: an exception was thrown, and we had to catch it and not let it escape.

How much of a problem is this? Not that much, really. First of all, releasing memory - the most frequently managed resource - does not throw. Usually, the memory is released not as just memory, but by deleting an object. But remember that the destructors should not throw, so the entire process of releasing the memory by deleting an object also does not throw. At this point, the reader might, in search of a counter-example, look up in the standard what happens if unlocking a mutex fails (that would force the destructor of `std::lock_guard` to deal with the error). The answer is both surprising and enlightening: unlocking a mutex cannot throw, but if it fails, undefined behavior results instead. This is no accident, the mutex was intended to work with an RAII object. Such is, in general, the C++ approach to releasing the resources: an exception should not be thrown if the release fails, or at least not allowed to propagate. It can be caught and logged, for example, but the calling program will, in general, remain unaware of the failure, possibly at the cost of undefined behavior.

Summary

We have introduced the most widely used idiom for resource management in C++, the "resource acquisition is initialization" idiom (RAII). With this idiom, each resource is owned by an object. Constructing (initializing) the object acquires the resource, and deleting the object releases it. We have seen how using RAII addresses the problems of resource management, such as leaking resources, accidentally sharing resources, and releasing resources incorrectly. Writing RAII objects is simple enough, but there are several caveats to keep in mind. Finally, we reviewed the complications that arise when error handling has to be combined with RAII.

Questions

1. What are the "resources" that a program can manage?
2. What are the main considerations when managing resources in a C++ program?
3. What is the RAII?
4. How does RAII address the problem of leaking resources?
5. How does RAII address the problem of dangling resource handles?
6. What RAII objects are provided by the C++ standard library?
7. What precautions must be taken when writing RAII objects?
8. What happens if releasing a resource fails?

Further Reading

- https://www.packtpub.com/application-development/expert-c-programming
- https://www.packtpub.com/application-development/c-data-structures-and-algorithms
- https://www.packtpub.com/application-development/rapid-c-video

TBD - OOP Idiom

Coming soon...

Type Erasure

Coming soon...

SFINAE and Overload Resolution Management

Coming soon...

Classic Patterns for Object-Oriented Designs

Coming soon...

Local Buffer Optimization

Not all design patterns are concerned with designing class hierarchies. For commonly occurring problem, a software design pattern is the most general and reusable solution, and, for those programming in C++, one of the most commonly occurring problems is inadequate performance. Among those, the problems of efficient memory management are some of the most frequent. Perforce, few general solutions, that is patterns. Patterns were developed to deal with these problems. In this chapter, we will explore one such pattern that addresses, in particular, the overhead of small, frequent memory allocations.

The following topics will be covered in this chapter:

- What is the overhead of small memory allocations, and how to measure it?
- What is the local buffer optimization, how does it improve performance, and how the improvements can be measured?
- When can the local buffer optimization pattern be used effectively?
- What are the possible downsides of and restrictions on the use of the local buffer optimization pattern?

Technical requirements

Google Benchmark library: `https://github.com/google/benchmark`.

Example code: `https://github.com/PacktPublishing/Hands-On-Design-Patterns-with-C-/tree/master/Ch10`

The overhead of small memory allocations

The local buffer optimization is just that, an optimization. It is a performance-oriented pattern, and we must, perforce, keep in mind the first rule of performance—never guess about performance. Performance and the effect of any optimization must be measured.

Installing the micro-benchmark library

In our case, we are interested in the efficiency of memory allocations and small fragments of code that may contain such allocations. The appropriate tool for measuring the performance of small fragments of code is a micro-benchmark. There are many micro-benchmark libraries and tools out there; in this book, we will use the Google Benchmark library. To follow along with the examples in this chapter, you must first download and install the library (follow the instructions in the `Readme.md` file). Then you can compile and run the examples. You can build the sample files included with the library to see how to build a benchmark on your particular system. For example, on a Linux machine, the command to build and run a benchmark program `malloc1.C` might look something like this:

```
$CXX malloc1.C -I. -I$GBENCH_DIR/include -g -O4 -I. -Wall -Wextra -
Werror -pedantic --std=c++14 $GBENCH_DIR/lib/libbenchmark.a -
lpthread -lrt -lm -o malloc1 && ./malloc1
```

Here `$CXX` is your C++ compiler, such as `g++` or `g++-6`, and `$GBENCH_DIR` is the directory where the benchmark is installed.

The cost of memory allocations

Since we are exploring the overhead of memory allocations and the ways to reduce it, the first question we must answer is how expensive a memory allocation is. After all, nobody wants to optimize something so fast that it needs no optimization. We can use the Google Benchmark (or any other micro-benchmark, if you prefer) to answer this question. The simplest benchmark to measure the cost of memory allocation might look like this:

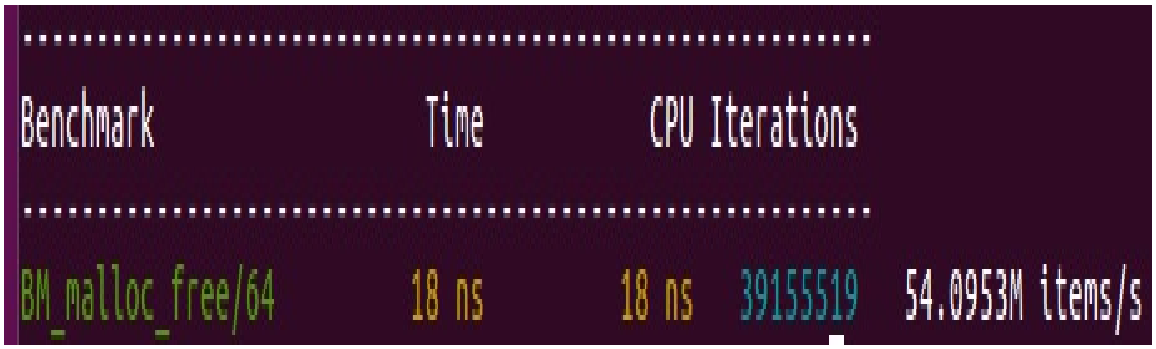
```
void BM_malloc(benchmark::State& state) {
    for (auto _ : state) {
        void* p = malloc(64);
        benchmark::DoNotOptimize(p);
    }
    state.SetItemsProcessed(state.iterations());
}
BENCHMARK(BM_malloc_free);
```

The `benchmark::DoNotOptimize` wrapper prevents the compiler from optimizing away the unused variable. Alas, this experiment is probably not going to end well—the micro-benchmark library needs to run the test many times, often millions of times, to accumulate accurate enough average run time. It is highly likely that the machine will run out of memory before the benchmark is complete. The fix is easy enough-we must also free the memory we allocated:

```
void BM_malloc_free(benchmark::State& state) {
    const size_t S = state.range(0);
    for (auto _ : state) {
        void* p = malloc(S);
        benchmark::DoNotOptimize(p);
```

```
        free(p);
    }
    state.SetItemsProcessed(state.iterations());
}
BENCHMARK(BM_malloc_free)->Arg(64);
```

One must note that we now measure the cost of both allocation and deallocation, which is reflected in the changed name of the function. This is not an unreasonable change—any allocated memory will need to be deallocated sometime later, so the cost must be paid at some point. We have also changed the benchmark to be parametrized by the allocation size. If you run this benchmark, you should get something like this:



This tells us that allocation and deallocation of 64 bytes of memory cost about 18 nanoseconds on this particular machine, which adds up to 54 million allocations/deallocations per second. If you're curious whether the 64 bytes size is special in some way, you can change the size value in the argument of the benchmark, or run the benchmark for a whole range of sizes:

```
void BM_malloc_free(benchmark::State& state) {
    const size_t S = state.range(0);
    for (auto _ : state) {
        void* p = malloc(S);
        benchmark::DoNotOptimize(p);
        free(p);
    }
    state.SetItemsProcessed(state.iterations());
}
```

```
BENCHMARK(BM_malloc_free)->RangeMultiplier(2)->Range(32, 256);
```

One might also note that, so far, we have measured the time it takes to make the very first memory allocation in the program since we have not allocated anything else. The C++ run-time system probably did some dynamic allocations at the startup of the program, but still, this is not a very realistic benchmark. We can make the measurement more relevant by pre-allocating some amount of memory:

```
#define REPEAT2(x) x x
#define REPEAT4(x) REPEAT2(x) REPEAT2(x)
#define REPEAT8(x) REPEAT4(x) REPEAT4(x)
#define REPEAT16(x) REPEAT8(x) REPEAT8(x)
#define REPEAT32(x) REPEAT16(x) REPEAT16(x)
#define REPEAT(x) REPEAT32(x)

void BM_malloc_free(benchmark::State& state) {
    const size_t S = state.range(0);
    const size_t N = state.range(1);
    std::vector<void*> v(N);
    for (size_t i = 0; i < N; ++i) v[i] = malloc(S);
    for (auto _ : state) {
        REPEAT({
        void* p = malloc(S);
        benchmark::DoNotOptimize(p);
        free(p);
        });
    }
    state.SetItemsProcessed(32*state.iterations());
    for (size_t i = 0; i < N; ++i) free(v[i]);
}
BENCHMARK(BM_malloc_free)->RangeMultiplier(2)->Ranges({{32, 256},
{1<<15, 1<<15}});
```

Here we make `N` calls to `malloc` before starting the benchmark. Further improvements can be achieved by varying the allocation size during the preallocations. We have also replicated the body of the benchmark loop 32 times (using the C preprocessor macro) to reduce the overhead of the loop itself on the

measurement. The time reported by the benchmark is now the time it takes to do 32 allocations and deallocations, which is not very convenient, but the allocation rate remains valid since we have accounted for the loop unrolling and multiplied the number of iterations by 32 when setting the number of processed items (in Google Benchmark, an item is whatever you want it to be, and the number of items per second is reported at the end of the benchmark, so we have declared one allocation/deallocation to be an item).

Even with all these modifications and improvements, the final result is going to be pretty close to our initial measurement of 54 million allocations per second. This seems very fast, just 18 nanoseconds. Remember, however, that a modern CPU can do dozens of instructions in this time. As we are dealing with small allocations, it is highly likely that the processing time spent on each allocated memory fragment is also small, and the overhead of allocation is non-trivial. This, of course, represents guessing about performance and something I warned you against, and so we will confirm this claim by direct experiments.

First, however, I want to show you another reason why small memory allocations are particularly inefficient. So far, we have explored the cost of memory allocations on only one thread. Today, most programs that have any performance requirements at all, are concurrent, and C++ supports concurrency and multi-threading. Let us take a look at how the cost of memory allocations changes when we do it on several threads at once:

```
void BM_malloc_free(benchmark::State& state) {
    const size_t S = state.range(0);
    const size_t N = state.range(1);
    std::vector<void*> v(N);
    for (size_t i = 0; i < N; ++i) v[i] = malloc(S);
    for (auto _ : state) {
        REPEAT({
```

```
        void* p = malloc(S);
        benchmark::DoNotOptimize(p);
        free(p);
        });
    }
    state.SetItemsProcessed(32*state.iterations());
    for (size_t i = 0; i < N; ++i) free(v[i]);
}
BENCHMARK(BM_malloc_free)->RangeMultiplier(2)->Ranges({{32, 256},
{1<<15, 1<<15}})->ThreadRange(1, 2);
```

The result greatly depends on the hardware and the version of `malloc` used by the system. Also, on large machines with many CPUs, you can do much more than two threads. Nonetheless, the overall trend should look something like this:

```
-------------------------------------------------------------------------
Benchmark                               Time           CPU Iterations
-------------------------------------------------------------------------
BM_malloc_free/32/32768/threads:1      649 ns        648 ns    1044207   47.1165M items/s
BM_malloc_free/32/32768/threads:2     1077 ns       2153 ns     327732   14.1718M items/s
```

This is quite dismal—the cost of allocations increased several times when we went from one thread to two (on a larger machine, a similar increase is going to happen but probably with more than two threads). The system memory allocator appears to be a bane of effective concurrency. There are better allocators that can be used to replace the default `malloc()`, but they do have their own downsides. Plus, it would be better if our C++ program did not depend on a particular, non-standard, system library replacement for its performance. We need a better way to allocate memory. Let us have a look at it.

Introducing the local buffer optimization

The least amount of work a program can do to accomplish a certain task is no work at all. Free stuff is great. Similarly, the fastest way to allocate and deallocate memory is this—don't. The local buffer optimization is a way to get something for nothing, in this case, to get some memory for no additional computing cost.

The main idea

To understand the local buffer optimization, you have to remember that memory allocations do not happen in isolation. Usually, if a small amount of memory is needed, the allocated memory is used as a part of some data structure. For example, let us consider a very simple character string:

```
class simple_string {
    public:
    simple_string() : s_() {}
    explicit simple_string(const char* s) : s_(strdup(s)) {}
    simple_string(const simple_string& s) : s_(strdup(s.s_)) {}
    simple_string& operator=(const char* s) { free(s_); s_ =
strdup(s); return *this; }
    simple_string& operator=(const simple_string& s) { free(s_); s_
= strdup(s.s_); return *this; }
    bool operator==(const simple_string& rhs) const { return
strcmp(s_, rhs.s_) == 0; }
    ~simple_string() { free(s_); }
    private:
    char* s_;
};
```

The string allocates its memory from `malloc()` via a `strdup()` call and returns it by calling `free()`. To be in any way useful, the string would need many more member functions, but these are sufficient for now to explore the overhead of memory allocation. Speaking of allocation, every time a string is constructed, copied, or assigned, an allocation happens. To be more precise, every time a string is constructed, an additional allocation happens: the string object itself has to be allocated somewhere, which may be on the stack for a local variable, or on the heap if the string is a part of some dynamically allocated data structure. In addition

to that, an allocation for the string data happens, and the memory is always taken from `malloc()`.

This, then, is the idea of the local buffer optimization-why don't we make the string object larger so it can contain its own data? That would really be getting something for nothing—the memory for the string object has to be allocated anyway, but the additional memory for the string data we would get at no extra cost. Of course, a string can be arbitrarily long, and so we do not know in advance how much larger do we need to make the string object to store any string the program will encounter. Even if we did, it would be a tremendous waste of memory to always allocate the object of that large size, even for very short strings. We can, however, make an observation—the longer the string is, the longer it takes to process it (copy, search, convert, or whatever we need to do with it). For very long strings, the cost of allocations is going to be small compared to the cost of processing. For short strings, on the other hand, the cost of the allocation could be significant. Therefore, the most performance benefit can be obtained by storing short strings in the object itself, while any string that is too long to fit in the object will be stored in allocated memory, as before. This is, in a nutshell, the local buffer optimization, which, for strings, is also known as a **short string optimization**—the object (string) contains a local buffer of a certain size, and any string that fits into that buffer is stored directly inside the object:

```
class small_string {
    public:
    small_string() : s_() {}
    explicit small_string(const char* s)
        : s_((strlen(s) + 1 < sizeof(buf_)) ? buf_ : strdup(s))
    {
        if (s_ == buf_) strncpy(buf_, s, sizeof(buf_));
    }
    small_string(const small_string& s)
        : s_((s.s_ == s.buf_) ? buf_ : strdup(s.s_))
```

```
    {
        if (s_ == buf_) memcpy(buf_, s.buf_, sizeof(buf_));
    }
    small_string& operator=(const char* s) {
        if (s_ != buf_) free(s_);
        s_ = (strlen(s) + 1 < sizeof(buf_)) ? buf_ : strdup(s);
        if (s_ == buf_) strncpy(buf_, s, sizeof(buf_));
        return *this;
    }
    small_string& operator=(const small_string& s) {
        if (s_ != buf_) free(s_);
        s_ = (s.s_ == s.buf_) ? buf_ : strdup(s.s_);
        if (s_ == buf_) memcpy(buf_, s.buf_, sizeof(buf_));
        return *this;
    }
    bool operator==(const small_string& rhs) const {
       return strcmp(s_, rhs.s_) == 0;
    }
    ~small_string() {
       if (s_ != buf_) free(s_);
    }
    private:
    char* s_;
    char buf_[16];
};
```

In this code example, the buffer size is set statically at 16 characters, including the null character used to terminate the string. Any string that is longer than 16 is allocated from `malloc()`. When assigning or destroying a string object, one must check its effect whether the allocation was done or the internal buffer was used, to appropriately release the memory used by the string.

Effect of the local buffer optimization

How much faster is the `small_string` compared to the `simple_string`? That depends, of course, on what you need to do with it. Let's start with just creating and deleting the strings. To avoid typing the same benchmark code twice, we can use the template benchmark:

```
template <typename T>
void BM_string_create_short(benchmark::State& state) {
    const char* s = "Simple string";
    for (auto _ : state) {
        REPEAT({ T S(s); benchmark::DoNotOptimize(S); })
    }
    state.SetItemsProcessed(32*state.iterations());
}
BENCHMARK_TEMPLATE1(BM_string_create_short, simple_string);
BENCHMARK_TEMPLATE1(BM_string_create_short, small_string);
```

The result is quite impressive:

```
-------------------------------------------------------------------------------
Benchmark                                     Time           CPU Iterations
-------------------------------------------------------------------------------
BM_string_create_short<simple_string>       726 ns        726 ns     928017   42.0364M items/s
BM_string_create_short<small_string>         29 ns         29 ns   24449910   1067.15M items/s
```

It gets even better when we try the same test on multiple threads:

```
-------------------------------------------------------------------------------
Benchmark                                        Time           CPU Iterations
-------------------------------------------------------------------------------
BM_string_create_short<simple_string>/threads:2      1117 ns       2233 ns     312608   13.6656M items/s
BM_string_create_short<small_string>/threads:2         15 ns         31 ns   22646526    997.64M items/s
```

While the regular string creation is much slower on multiple threads, the small string shows no such penalty and, in fact, scales almost perfectly. Of course, this is pretty much the best case scenario for the small string optimization—first, all we do is create and delete strings, which is the very part we optimized, and second, the string is a local variable, its memory is allocated as a part of the stack frame so there is no additional allocation cost. However, this is not an unreasonable case—after all, local variables are not rare at all, and, if the string is a part of some larger data structure, the allocation cost for that structure has to be paid anyway, so allocating anything else at the same time and without additional cost is effectively free.

Nonetheless, it is unlikely that we only allocate the strings to immediately deallocate them, so we should consider the cost of other operations. We can expect similar improvements for copying or assigning strings, as long as they stay short, of course:

```
template <typename T>
void BM_string_copy_short(benchmark::State& state) {
    const T s("Simple string");
    for (auto _ : state) {
        REPEAT({ T S(s); benchmark::DoNotOptimize(S); })
    }
    state.SetItemsProcessed(32*state.iterations());
}

template <typename T>
void BM_string_assign_short(benchmark::State& state) {
    const T s("Simple string");
    T S;
    for (auto _ : state) {
        REPEAT({ benchmark::DoNotOptimize(S = s); })
    }
```

```
    state.SetItemsProcessed(32*state.iterations());
}

BENCHMARK_TEMPLATE1(BM_string_copy_short, simple_string);
BENCHMARK_TEMPLATE1(BM_string_copy_short, small_string);
BENCHMARK_TEMPLATE1(BM_string_assign_short, simple_string);
BENCHMARK_TEMPLATE1(BM_string_assign_short, small_string);
```

Indeed, the similar dramatic performance gain is observed:

```
-------------------------------------------------------------------------------
Benchmark                                        Time          CPU Iterations
-------------------------------------------------------------------------------
BM_string_copy_short<simple_string>            706 ns       706 ns     971142   43.2178M items/s
BM_string_copy_short<small_string>              45 ns        45 ns   15621788   672.256M items/s
BM_string_assign_short<simple_string>          754 ns       753 ns     924908   40.5313M items/s
BM_string_assign_short<small_string>            51 ns        51 ns   13281364   596.836M items/s
```

We are likely also to need to read the data in the strings, at least once, to compare them or search for a specific string or character, or compute some derived value. We do not expect improvements of the similar scale for these operations, of course, since none of them involves any allocations or deallocations. One might ask why, then, should we expect any improvements at all? Indeed, a simple test of string comparison, for example, shows no difference between the two versions of the string. In order to see any benefit, we have to create many string objects and compare them:

```
template <typename T>
void BM_string_compare_short(benchmark::State& state) {
    const size_t N = state.range(0);
    const T s("Simple string");
    std::vector<T> v1, v2;
   ... populate the vectors with strings ...
    for (auto _ : state) {
        for (size_t i = 0; i < N; ++i)
benchmark::DoNotOptimize(v1[i] == v2[i]);
    }
    state.SetItemsProcessed(N*state.iterations());
}
```

```
#define ARG Arg(1<<22)
BENCHMARK_TEMPLATE1(BM_string_compare_short, simple_string)->ARG;
BENCHMARK_TEMPLATE1(BM_string_compare_short, small_string)->ARG;
```

For small values of `N` (small total number of strings), there won't be any significant benefit from the optimization. But when we have to process many strings, comparing strings with the small string optimization can be approximately twice as fast:

```
-------------------------------------------------------------------------------
Benchmark                                        Time          CPU Iterations
-------------------------------------------------------------------------------
BM_string_compare_short<simple_string>/4194304    36388678 ns   36385972 ns        19   109.932M items/s
BM_string_compare_short<small_string>/4194304     14640789 ns   14639702 ns        55    273.23M items/s
```

Why is that happening, if there are no allocations at all? This experiment shows the second, very important benefit of the local buffer optimization: improved cache locality. The string object itself has to be accessed before the string data can be read—it contains the pointer to the data. For the regular string, accessing the string characters involves two memory accesses at different, generally unrelated addresses. If the total amount of data is large, the second access, to the string data, is likely to miss the cache and wait for the data to be brought from the main memory. On the other hand, the optimized string keeps the data close to the string object, so, once the string itself is in the cache, so is the data. The reason that we need enough different strings to see this benefit is that, with few strings, all string objects and their data can reside in the cache permanently. Only when the total size of the strings exceeds the size of the cache will the performance benefits manifest. Now, let us dive deeper with some additional optimizations.

Additional optimizations

The `simple_string` class we have implemented has an obvious inefficiency—when the string is stored in the local buffer, we do not really need the pointer to the data. We know exactly where the data is, in the local buffer. We do need to know, somehow, whether the data is in the local buffer or in the externally allocated memory, but we don't need to use 8 bytes (on a 64-bit machine) just to store that. Of course, we still need the pointer for storing longer strings, but we could reuse that memory for the buffer when the string is short:

```
class small_string {
    ...
    private:
    union {
    char* s_;
    struct {
        char buf[15];
        char tag;
    } b_;
    };
};
```

Here we use the last byte as a tag to indicate whether the string is stored locally (`tag == 0`) or in a separate allocation (`tag == 1`). Note that the total buffer size is still 16 characters, 15 for the string itself and one for the tag, which also doubles at the trailing zero if the string needs all 16 bytes (this is why we have to use `tag == 0` to indicate local storage, it would cost us an extra byte to do otherwise). The pointer is overlaid in memory with the first 8 bytes of the character buffer. In this example, we have chosen to optimize the total memory occupied by the string: this string still

has a 16-character local buffer, just like the previous version, but the object itself is now only 16 bytes, not 24. If we were willing to keep the object size the same, we could have used a larger buffer and stored longer strings locally. The benefit of the small string optimization does, generally, diminish as the strings become longer. The optimal crossover point from local to remote strings depends on the particular application and must, of course, be determined by benchmark measurements.

Local buffer optimization beyond strings

The local buffer optimization can be used effectively for much more than just short strings. In fact, any time a small dynamic allocation of a size that is determined at run-time is needed, this optimization should be considered. In this section, we will consider several such data structures.

Small vector

Another very common data structure that often benefits from the local buffer optimization is a vector. Vectors are essentially dynamic contiguous arrays of data elements of the specified type (in this sense, a string is a vector of bytes, although the null termination gives strings their own specifics). A basic vector, such as the `std::vector` found in the C++ standard library, needs two data members, data pointer, and data size:

```
class simple_vector {
    public:
    simple_vector() : n_(), p_() {}
    simple_vector(std::initializer_list<int> il)
       : n_(il.size()), p_(static_cast<int*>
(malloc(sizeof(int)*n_)))
    {
        int* p = p_;
        for (auto x : il) *p++ = x;
    }
    ~simple_vector() {
        free(p_);
    }
    size_t size() const { return n_; }
    private:
    size_t n_;
    int* p_;
};
```

Vectors are usually templates, like the standard `std::vector`, but we have simplified this example to show a vector of integers (converting this vector class to a template is left as an exercise to the reader and does not in any way alter the application of the local buffer optimization pattern). We can apply the *small vector*

optimization and store the vector data in the body of the vector object as long as it is small enough:

```
class small_vector {
    public:
    small_vector() : n_(), p_() {}
    small_vector(std::initializer_list<int> il)
        : n_(il.size()),
          p_((n_ < sizeof(buf_)/sizeof(buf_[0])) ? buf_ :
static_cast<int*>(malloc(sizeof(int)*n_)))
    {
        int* p = p_;
        for (auto x : il) *p++ = x;
    }
    ~small_vector() {
        if (p_ != buf_) free(p_);
    }
    private:
    size_t n_;
    int* p_;
    int buf_[16];
};
```

We can further optimize the vector, similarly to the string, and overlay the local buffer with the pointer. We cannot use the last byte as a tag, as we did before, since any element of the vector can have any value, and the value of zero is, in general, not special. However, we need to store the size of the vector anyway, so we can use it at any time to determine whether the local buffer is used or not. We can further take advantage of the fact that, if the local buffer optimization is used, the size of the vector cannot be very large, so we do not need a field of type `size_t` to store it:

```
class small_vector {
    public:
    small_vector() { short_.n = 0; }
    small_vector(std::initializer_list<int> il) {
        int* p;
        if (il.size() < sizeof(short_.buf)/sizeof(short_.buf[0])) {
```

```
            short_.n = il.size();
            p = short_.buf;
        } else {
            short_.n = UCHAR_MAX;
            long_.n = il.size();
            p = long_.p = static_cast<int*>
(malloc(sizeof(int)*long_.n));
        }
        for (auto x : il) *p++ = x;
    }
    ~small_vector() {
        if (short_.n == UCHAR_MAX) free(long_.p);
    }
    private:
    union {
        struct {
            size_t n;
            int* p;
        } long_;
        struct {
            int buf[15];
            unsigned char n;
        } short_;
    };
};
```

Here we store the vector size either in `size_t long_.n` or in `unsigned char short_.n`, depending on whether or not the local buffer is used. A remote buffer is indicated by storing `UCHAR_MAX` (255) in the *short size*. Since this value is larger than the size of the local buffer, this *tag* is unambiguous (were the local buffer increased to store more than 255 elements, the type of `short_.n` would need to be changed to a longer integer.)

We can measure the performance gains from the small vector optimization using a benchmark similar to the one we used for the strings. Depending on the actual size of the vector, gains of about 10x in creating and copying the vectors can be expected, more if the benchmark runs on multiple threads. Of course, other data structures can be optimized in a similar manner when

they store small amounts of dynamically allocated data.

Type-erased and callable objects

There is another, very different, type of applications where the local buffer optimization can be used very effectively—storing callable objects, which is an object that can be invoked as a function. Many template classes provide an option to customize some part of their behavior using a callable object. For example, the C++ standard shared pointer `std::shared_ptr` allows the user to specify a custom deleter. This deleter will be called with the address of the object to be deleted, so it is a callable with one argument of type `void*`. It could be a function pointer, a member function pointer, or a functor object (an object with an `operator()` defined)—any type that can be called on a pointer `p`, that is, that compiles in the function call syntax `callable(p)` can be used. The deleter, however, is more than a type, it is an object and is specified at runtime, and so it needs to be stored someplace where the shared pointer can get to it. Were the deleter a part of the shared pointer type, we could simply declare a data member of that type in the shared pointer object (or, in case of the C++ shared pointer, in its reference object that is shared between all copies of the shared pointer). One could consider it a trivial application of the local buffer optimization, as in the following smart pointer that automatically deletes the object when the pointer goes out of scope (just like `std::unique_ptr`):

```
template <typename T, typename Deleter>
class smartptr {
    public:
    smartptr(T* p, Deleter d) : p_(p), d_(d) {}
    ~smartptr() { d_(p_); }
```

```
    T* operator->() { return p_; }
    const T* operator->() const { return p_; }
    private:
    T* p_;
    Deleter d_;
};
```

We are after more interesting things, however, and one such can be found when we deal with type-erased objects. The details of such objects were considered in the chapter dedicated to the type erasure, but, in a nutshell, they are objects where the callable is not a part of the type itself (is *erased* from the type of the containing object). The callable is instead stored in a polymorphic object, and a virtual function is used to call the object of the right type at run time. The polymorphic object, in turn, is manipulated through the base class pointer.

Now, we have a problem that is, in a sense, similar to the precedingsmall vector—we need to store some data, the callable object in our case, whose type, and, therefore, size, is not statically known. The general solution is to dynamically allocate such objects and access them through the base class pointer. In case of a smart pointer deleter, we could do it like this:

```
template <typename T>
class smartptr_te {
    struct deleter_base {
        virtual void apply(void*) = 0;
        virtual ~deleter_base() {}
    };
    template <typename Deleter> struct deleter : public
deleter_base {
        deleter(Deleter d) : d_(d) {}
        virtual void apply(void* p) { d_(static_cast<T*>(p)); }
        Deleter d_;
    };
    public:
    template <typename Deleter> smartptr_te(T* p, Deleter d)
        : p_(p), d_(new deleter<Deleter>(d))
```

```
    {}
    ~smartptr_te() { d_->apply(p_); delete d_; }
    T* operator->() { return p_; }
    const T* operator->() const { return p_; }
    private:
    T* p_;
    deleter_base* d_;
};
```

Note that the `Deleter` type is no longer a part of the smart pointer type, it was *erased*. All smart pointers for the same object type `T` have the same type, `smartptr_te<T>` (here **te** stands for **type-erased**). However, we have to pay a steep price for this syntactic convenience—every time a smart pointer is created, there is an additional memory allocation. How steep? The first rule of performance must again be remembered—*steep* is only a guess until confirmed by an experiment, such as the following benchmark:

```
struct deleter {    // Very simple deleter, matches operator new
    template <typename T> void operator()(T* p) { delete p; }
};

void BM_smartptr(benchmark::State& state) {
    deleter d;
    for (auto _ : state) {
        smartptr<int, deleter> p(new int, d);
    }
    state.SetItemsProcessed(state.iterations());
}

void BM_smartptr_te(benchmark::State& state) {
    deleter d;
    for (auto _ : state) {
        smartptr_te<int> p(new int, d);
    }
    state.SetItemsProcessed(state.iterations());
}
BENCHMARK(BM_smartptr);
BENCHMARK(BM_smartptr_te);
```

For the smart pointer with statically defined deleter, we can expect the cost of each iteration to be very similar to the cost of calling `malloc()` and `free()` that we have measured earlier:

```
-----------------------------------------------------------
Benchmark                 Time           CPU Iterations
-----------------------------------------------------------
BM_smartptr              21 ns         21 ns   34069972   45.7846M items/s
BM_smartptr_te           42 ns         42 ns   16832659   22.7958M items/s
```

For the type-erased smart pointer, there are two allocations instead of one, and so the time it takes to create the pointer object is doubled. By the way, we can also measure the performance of a raw pointer, and it should be the same as the smart pointer within the accuracy of the measurements (that was, in fact, a stated design goal for the standard `std::unique_ptr`).

We can apply the same idea of the local buffer optimization here, and it is likely to be even more effective than for strings—after all, most callable objects are small. We can't completely count on that, however, and must handle the case of a callable object that is larger than the local buffer:

```
template <typename T>
class smartptr_te_lb {
    struct deleter_base {
        virtual void apply(void*) = 0;
        virtual ~deleter_base() {}
    };
    template <typename Deleter> struct deleter : public
deleter_base {
        deleter(Deleter d) : d_(d) {}
        virtual void apply(void* p) { d_(static_cast<T*>(p)); }
        Deleter d_;
    };
    public:
    template <typename Deleter> smartptr_te_lb(T* p, Deleter d) :
        p_(p),
        d_((sizeof(Deleter) > sizeof(buf_)) ? new deleter<Deleter>
(d) :
```

```
                                                    new (buf_)
deleter<Deleter>(d))
    {}
    ~smartptr_te_lb() {
        d_->apply(p_);
        if (static_cast<void*>(d_) == static_cast<void*>(buf_)) {
            d_->~deleter_base();
        } else {
            delete d_;
        }
    }
    T* operator->() { return p_; }
    const T* operator->() const { return p_; }
    private:
    T* p_;
    deleter_base* d_;
    char buf_[16];
};
```

Using the same benchmark as before, we can measure the performance of the type-erased smart pointer with the local buffer optimization. While construction and deletion of a smart pointer without type erasure took 21 nanoseconds, and 42 nanoseconds with type erasure, the optimized type-erased shared pointer test takes 23 nanoseconds on the same machine. The slight overhead comes from checking whether the deleter is stored locally or remotely.

Local buffer optimization in the C++ library

We should note that the last application of the local buffer optimization, storing callables for type-erased objects, is widely used in the C++ standard template library. For example, `std::shared_ptr` has a type-erased deleter, and most implementations use the local buffer optimization—the deleter is stored with the reference object and not with each copy of the shared pointer, of course. The `std::unique_pointer`, on the other hand, is not type-erased at all, to avoid even the small overhead, or, potentially, a much larger overhead if the deleter does not fit into the local buffer.

The *ultimate type-erased* object of the C++ standard library, `std::function`, is also typically implemented with a local buffer for storing small callable objects without the expense of an additional allocation. The universal container object for any type, `std::any` (since C++17) is also typically implemented without a dynamic allocation when possible.

Downsides of the local buffer optimization

The local buffer optimization is not without its downsides. The most obvious one is that all objects with a local buffer are larger than they would be without one. If the typical data stored in the buffer is smaller than the chosen buffer size, every object is wasting some memory, but at least the optimization is paying off. Worse, if our choice of the buffer size is badly off and most data is, in fact, larger than the local buffer, the data is stored remotely but the local buffers are still created inside every object, and all that memory is wasted. There is an obvious trade-off between the amount of memory we are willing to waste and the range of data sizes where the optimization is effective. The size of the local buffer should be carefully chosen with the application in mind.

The more subtle complication is this: the data that used to be external to the object is now stored inside the object. This has several consequences, in addition to the performance benefits we were so focused on. First of all, every copy of the object contains its own copy of the data as long as it fits into the local buffer. This prevents such designs as reference-counting of data; for example, a **copy-on-write** (**COW**) string, where the data is not copied as long as all string copies remain the same, cannot use the small string optimization. Second, the data must be moved if the object itself is moved. Contrast this with the `std::vector`, which is moved or swapped, essentially, like a pointer—the pointer to the data is moved but the data remains in place. Similar consideration exists for the object contained inside `std::any`. One could dismiss this concern as trivial—after all, the local buffer

optimization is used primarily for small amounts of data, and the cost of moving them should be comparable to the cost of copying the pointer. However, more than performance is at stake here: moving an `std::vector` (or an `std::any`, for that matter) is guaranteed not to throw an exception. Moving an arbitrary object, in general, offers no such guarantees. Therefore, `std::any` can be implemented with a local buffer optimization only if the object it contains is `std::is_nothrow_move_constructible`. Even such guarantee does not suffice for the case of `std::vector`, however—the standard explicitly states that moving, or swapping, a vector does not invalidate iterators pointing to any element of the vector. Obviously, this requirement is incompatible with the local buffer optimization, since moving a small vector would relocate all its elements to a different region of memory. For that reason, many high-efficiency libraries offer a custom vector-like container that supports small vector optimization at the expense of the standard iterator invalidation guarantees.

Summary

We have just introduced a design pattern which is aimed solely at improved performance. Efficiency is an important consideration for C++ language, thus, the C++ community developed patterns to address the most common inefficiencies. Repeated or wasteful memory allocation is, perhaps, the most common of all. The design pattern we have just seen, the local buffer optimization, is a powerful tool that can greatly reduce such allocations. We have seen how it can be applied to compact data structures, as well as to store small objects such as callables. We have also reviewed possible downsides of using this pattern.

Questions

1. How can we measure the performance of a small fragment of code?
2. Why are small and frequent memory allocations particularly bad for performance?
3. What is the local buffer optimization, and how does it work?
4. Why is an allocation of an additional buffer inside an object effectively *free*?
5. What is a short string optimization?
6. What is a small vector optimization?
7. Why is the local buffer optimization particularly effective for callable objects?
8. What are the tradeoffs to consider when using local buffer optimization?
9. When should an object not be placed in a local buffer?

Further reading

- *C++ High Performance* by Viktor Sehr, Björn Andrist:

  https://www.packtpub.com/application-development/c-high-performance

- *C++ Data Structures and Algorithms* by Wisnu Anggoro:

  https://www.packtpub.com/application-development/c-data-structures-and-algorithms

- *High Performance Applications with C++* [Video] by Jeganathan Swaminathan:

  https://www.packtpub.com/application-development/high-performance-applications-c-video

Friend Factory

Coming soon...

Virtual Constructors and Factories

Coming soon...

Memory Ownership

Memory mismanagement is one of the most common problems in C++ programs. Many of these problems boil down to incorrect assumptions about which part of the code or which entity owns a particular memory. Then we get memory leaks, accessing unallocated memory, excessive memory use, and other difficult to debug problems. Modern C++ has a set of memory ownership idioms, which, taken together, allow the programmer to clearly express the design intent when it comes to memory ownership. This, in turn, makes it much easier to write code that correctly allocates, accesses and deallocates memory.

The following topics are covered in this chapter:

- What is memory ownership?
- What are the characteristics of a well-designed resource ownership?
- When and how to be agnostic about resource ownership?
- How to express exclusive memory ownership in C++?
- How to express shared memory ownership in C++?
- What is the cost of different memory ownership language constructs?

Technical requirements

C++ Core Guidelines: `https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md`

C++ Guideline Support library (GSL): `https://github.com/Microsoft/GSL`

What is memory ownership

In C++, the term **memory ownership** refers to the entity that is responsible for enforcing the lifetime of a particular memory allocation. In reality, we rarely talk about the ownership of raw memory. Usually, we manage the ownership and the lifetime of the objects that reside in said memory, and memory ownership is really just a shorthand for *object ownership*. The concept of memory ownership is closely tied to that of *resource ownership*. First of all, memory is a resource. It is not the only resource a program can manage, but it is by far the most commonly used one. Second, the C++ way of managing the resources is to have objects own them. Thus, the problem of managing resources is reduced to the problem of managing the owning objects, which, as we just learned, is what we really mean when we talk about memory ownership. In this context, memory ownership is about owning more than the memory, and mismanaged ownership can leak, miscount, or lose track of any resource that can be controlled by the program—memory, mutexes, files, database handles, cat videos, airline seat reservations, or nuclear warheads.

Well-designed memory ownership

How does well-designed memory ownership look like? The naive answer that first comes up is that at every point in the program it is clear who owns what object. This, however, is overly constraining—most of the program does not deal with ownership of resources, including memory. These parts of the program merely use resources. When writing such code, it is sufficient to know that a particular function or class does not own the memory. It is completely unimportant to know who does:

```
struct MyValues { long a, b, c, d; }
void Reset(MyValues* v) {    // Don't care who owns v, as long as
we don't
    v->a = v->b = v->c = v->d = 0;
}
```

How about this, then—at every point in the program it is clear who owns that object, or it is clear that the ownership is not changing? This is better, most of the code will fall under the second part of the answer. However, it's still too constraining—when taking ownership of an object, it is usually not important to know who is it taken from:

```
 class A {
     public:
     A(std::vector<int>&& v) : v_(std::move(v)) {}   // Transfers
ownership from whomever
     private:
     std::vector<int> v_;                            // We own this
now
```

```
};
```

Similarly, the whole point of the shared ownership (expressed through the reference-counted `std::shared_ptr`) is that we don't need to know who else owns the object:

```
class A {
    public:
    A(std::shared_ptr<std::vector<int>> v) : v_(v) {} // No idea
who owns v, don't care
    private:
    std::shared_ptr<std::vector<int>> v_;    // Sharing ownership
with any number of owners
};
```

A more accurate description of well-designed memory ownership takes more than one quoted sentence. Generally, the following are the attributes of good memory ownership practices:

- If a function or a class does not alter the memory ownership in any way, this should be clear to every client of this function or class, as well as the implementer.
- If a function or a class takes exclusive ownership of some of the objects passed to it, this should be clear to the client (we assume that the implementer knows it already, since he has to write the code).
- If a function or a class may share ownership of an object passed to it, this should be clear to the client (or anyone who reads the client code, for that matter).
- For every object that is created, at every point that it's used, it is clear whether this code is expected to delete it,

or not.

Poorly designed memory ownership

Just like good memory ownership defies a simple description and instead is characterized by a set of criteria it satisfies, so can bad memory ownership practices be recognized by their common manifestations. In general, where good design makes it clear whether the particular piece of code owns a resource or not, a bad design requires additional knowledge that cannot be deduced from the context. For example, who owns the object returned by this `MakeWidget()` function:

```
Widget* w = MakeWidget();
```

Is the client expected to delete the widget when it's no longer needed? If yes, how should it be deleted? If we decide to delete the widget and do it the wrong way, for example, by calling `operator delete` on a widget that was not, in fact, allocated by `operator new`, memory corruption will certainly result. Best case scenario, the program will just crash.

```
WidgetFactory WF;
Widget* w = WF.MakeAnother();
```

Does the factory own the widgets it created? Will it delete them when the factory object is deleted? Alternatively, is the client expected to do that? If we decide that the factory probably knows what it created, and will delete all such objects in due time, we may end up with a memory leak (or worse, if the objects owned some other resources).

```
Widget* w = MakeWidget();
Widget* w1 = Transmogrify(w);
```

Does `Transmogrify()` take ownership of the widget? Is the widget `w` still around after `Transmogrify()` is done with it? If the widget is deleted to construct the new, transmogrified, widget `w1`, we now have a dangling pointer. If the widget is not deleted, but we assume it might be, we have a memory leak.

Lest the reader thinks that all bad memory management practices can be recognized by the presence of raw pointers somewhere, here is an example of a rather poor approach to memory management that often arises as a knee-jerk response to the problems caused by the use of raw pointers:

```
void Double(std::shared_ptr<std::vector<int>> v) {
    for (auto& x : *v) {
        x *= 2;
    }
};
    ...
std::shared_ptr<std::vector<int>> v(...);
Double(v);
    ...
```

The function `Double()` is claiming in its interface that it takes shared ownership of the vector. However, that ownership is entirely gratuitous—there is no reason for `Double()` to own its argument, it does not attempt to extend its lifetime, it does not transfer ownership to anyone else, it merely modifies a vector passed in by the caller. We can reasonably expect that the caller owns the vector (or somebody else even higher in the call stack does), and that the vector will still be around when `Double()` returns control to the caller—after all, the caller wanted us to double the elements, presumably so he can do something else with them.

While this list is hardly complete, it serves to demonstrate the spectrum of the problems that can be caused by a slapdash approach to memory ownership.

Expressing memory ownership in C++

Throughout its history, the C++ language evolved in its approach to expressing memory ownership. The same syntactic constructs have been, at times, imbued with different assumed semantics. This evolution was partially driven by the new features added to the language (it's hard to talk about shared memory ownership if you don't have any shared pointers). On the other hand, most of the memory management tools added in C++11 and later were not new ideas or new concepts. The notion of a shared pointer has been around for a long time. The language support makes it easier to implement one (and having one in the standard library makes most custom implementations unnecessary), but shared pointers were used in C++ long before C++11 added them to the standard. The more important change that has occurred was the evolution of the understanding of the C++ community, the emergence of common practices and idioms. It is in this sense, as a set of conventions and semantics commonly associated with different syntactic features, that we can talk about the set of memory management practices as a design pattern of the C++ language. Let us now learn the ways to express different types of memory ownership.

Expressing non-ownership

Let's start with the most common kind of memory ownership—none. Most code does not allocate, deallocate, construct, or delete. It just does its work on objects that were created by someone else earlier and will be deleted by someone else later. How do you express the notion that a function is going to operate on an object but will not attempt to delete it or, conversely, extend its lifetime past the completion of the function itself? Very easy, and every C++ programmer have done it many times:

```
void Transmogrify(Widged* w) {        // I will not delete w
    ...
}
void MustTransmogrify(Widget& w) {    // Neither will I
    ...
}
class WidgetProcessor {
    public:
    WidgetProcessor(Widget* w) : w_(w) {}
    ...
    Widget* w_;                        // I do not own w
};
```

Non-owning access to an object should be granted using raw pointers or references. Yes, even in C++17, with all its smart pointers, there is a place for raw pointers. Not only that but in the bulk of the code, the majority of pointers will be raw pointers—all the non-owning ones.

The reader might reasonably point out at this time that the above example of the recommended practices for granting non-owning access looks exactly like one of the examples of bad practices

shown earlier. The distinction is in the context—in a well-designed program, only non-owning access is granted through raw pointers and references. Actual ownership is always expressed in some other way. Thus, it is clear that, when a raw pointer is encountered, the function or class is not going to mess with the ownership of the object in any way. This, of course, creates some confusion when it comes to converting old legacy code, with raw pointers everywhere, to the modern practices. As a matter of clarity, it is recommended to convert such code in parts, with clearly indicated transitions between the code that follows the modern guidelines and the one that does not.

Another issue to discuss here is the use of pointers versus references. As a matter of syntax, the reference is basically a pointer that cannot be `NULL` (or `nullptr`) and cannot be left uninitialized. It is tempting to adopt a convention that any pointer passed to a function may be `NULL` and must, therefore, be checked, and any function that cannot accept a `NULL` pointer must instead take a reference. It is a good convention, and widely used, but not widely enough to be considered an accepted design pattern. Perhaps in recognition of this, the C++ Core Guidelines library offers an alternative for expressing non-`NULL` pointers: `not_null<T*>`. Note that this is not a part of the language itself, but can be implemented in standard C++ without any language extension.

Expressing exclusive ownership

The second most common type of ownership is exclusive ownership—the code creates an object and will delete it later. The task of deletion will not be delegated to someone else, neither an extension of the lifetime of the object is permitted. This type of memory ownership is so common that we do it all the time without even thinking about it:

```
void Work() {
    Widget w;
    Transmogrify(w);
    Draw(w);
}
```

All local (stack) variables express unique memory ownership! Note that ownership in this context does not mean that someone else will not modify the object. It merely means that when the creator of the widget `w` - the function `DoWork()` in our case—decides to delete it, the deletion will succeed (nobody deleted it already) and the object will actually be deleted (nobody attempted to keep the object alive after the end of its scope).

This is the oldest way to construct an object in C++, and it's still the best one. If a stack variable does what you need, use it. C++11 provides another way to express unique ownership, and it is mainly used in cases where an object cannot be created on the stack but must be allocated on the heap. Heap allocation happens often when the ownership is shared or transferred: after all, the stack-allocated object will be deleted at the end of the containing scope, there is no way around it. If we need to keep the object alive longer, it has to be allocated somewhere else. But

we are talking about exclusive ownership here, the one that is not shared or given away. The other reason to create objects on the heap is that the size or type of the object may not be known at compile time. This usually happens when the object is polymorphic—a derived object is created, but the base class pointer is used. We now have a way of expressing the exclusive ownership of such objects using `std::unique_ptr`:

```
class FancyWidget : public Widget { ... };
std::unique_ptr<Widget> w(new FancyWidget);
```

What if the way to create an object is more complex than just `operator new`, and we need a factory function? That is the type of ownership we will consider next.

Expressing transfer of exclusive ownership

In the preceding example, a new object was created and immediately bound to a unique pointer `std::unique_ptr`, which guarantees exclusive ownership. The client code looks exactly the same if the object is created by a factory:

```
std::unique_ptr<Widget> w(WidgetFactory());
```

But what should the factory function return? It could certainly return a raw pointer, `Widget*`. After all, that is what `new` returns. But this opens the way to incorrect use of the `WidgetFactory`—for example, instead of capturing the returned raw pointer in a unique pointer, we could pass it to a function like `Transmogrify` that takes a raw pointer because it does not deal with the ownership. Now nobody owns the widget, and it ends up as a memory leak. Ideally, the `WidgetFactory` would be written in a way that would force the caller to take ownership of the returned object.

What we need here is an ownership transfer—the `WidgetFactory` is certainly an exclusive owner of the object it constructs, but at some point, it needs to hand off that ownership to a new, also exclusive, owner. The code to do it is very simple:

```
std::unique_ptr<Widget> WidgetFactory() {
    Widget* new_w = new Widget;
        ...
    return std::unique_ptr<Widget>(new_w);
}
std::unique_ptr<Widget> w(WidgetFactory());
```

This works exactly the way we would like, but why? Doesn't the unique pointer provide exclusive ownership? The answer is, it does, but it is also a movable object. Moving the content of a unique pointer into another one transfers the ownership of the object; the original pointer is left in the moved-from state (its destruction will not delete any objects). What is so good about this idiom? It clearly expresses, and forces at compilation time, that the factory expects the caller to take exclusive (or shared) ownership of the object. For example, this code, which would have left the new widget with no owner, does not compile:

```
void Transmogrify(Widget* w);
Transmogrify(WidgetFactory());
```

So how do we call `Transmogrify()` on a widget, after we properly assumed ownership? Still with a raw pointer:

```
std::unique_ptr<Widget> w(WidgetFactory());
Transmogrify(&*w);    // or w.get() - same thing, gives non-owning
access
```

But, what about the stack variables? Can the exclusive ownership be transferred to someone else before the variable is destroyed? This is going to be slightly more complicated—the memory for the object is allocated on the stack and is going away, so some amount of copying is involved. Exactly how much copying depends on whether the object is movable. Moving, in general, transfers the ownership from the moved-from object to the moved-to one. This can be used for return values but is more often used for passing arguments to functions that take exclusive ownership. Such functions must be declared to take the parameters by `rvalue` reference `T&&`:

```
void Consume(Widget&& w) { auto my_w = std::move(w); ... }
Widget w, w1;
Consume(std::move(w));    // No more w - it's in moved-from state
now
Consume(w1);              // Does not compile - must consent to
move
```

Note that the caller must explicitly give up the ownership by wrapping the argument in `std::move`. This is one of the advantages of this idiom; without it, an ownership-transferring call would look exactly the same as a regular call.

Expressing shared ownership

The last type of ownership we need to cover is the shared ownership, where multiple entities own the object equally. First, a word of caution—shared ownership is often misused, or over-used. Consider the preceding example, where a function was passed a shared pointer to an object it did not need to own. It is tempting to let the reference counting deal with the ownership of objects and not worry about deletion. However, this often is a sign of a poor design. In most systems, at some level, there is a clear ownership of resources, and it should be reflected in the chosen design of resource management. *The not worry about deletion* concern remains valid, explicit deletion of objects should be rare, but automatic deletion does not require shared ownership, merely a clearly expressed one (unique pointers, data members, and containers provide automatic deletion just as well).

That being said, there are definite cases for shared ownership. The most common valid applications of shared ownership are at a low level, inside data structures such as lists, trees, and more. A data element may be owned by other nodes of the same data structure, by any number of iterators currently pointing to it, and, possibly, by some temporary variables inside data structure member functions that operate on the entire structure or a part of it (such as rebalancing a tree). The ownership of the entire data structure is usually clear in a well thought-out design. But the ownership of each node, or data element, may be truly shared in the sense that any owner is equal to any other, none is privileged or primary.

In C++, the notion of the shared ownership is expressed through a shared pointer, `std::shared_ptr`:

```
struct ListNode {
    T data;
    std::shared_ptr<ListNode> next, prev;
};
class ListIterator {
    ...
    std::shared_ptr<ListNode> node_p;
};
```

The advantage of this design is that a list element that was unlinked from the list remains alive for as long as there is a way to access it through an iterator. This is not the way `std::list` is done, and it does not provide such guarantees. However, it may be a valid design for certain applications, for example, for a thread-safe list. Note that this particular application would also require atomic shared pointers which are only available in C++20 (or you can write your own using C++11).

Now, what about functions taking shared pointers as parameters? In a program that follows good memory ownership practices, such function conveys to the caller that it intends to take partial ownership that lasts longer than the function call itself—a copy of the shared pointer will be created. In the concurrent context, it may also indicate that the function needs to protect the object from deletion by another thread for at least as long as it's executing.

There are several disadvantages to shared ownership that one must keep in mind. The best-known one is the bane of shared pointers, the circular dependency. If two objects with shared pointers point to each other, the entire pair remains *in use* indefinitely. C++ offers a solution to that in the form of `std::weak_ptr`, a counterpart to the shared pointer that provides a

safe pointer to an object that may have already been deleted. If the previously mentioned pair of objects uses one shared and one weak pointer, the circular dependency is broken.

The circular dependency problem is real, but it happens more often in designs where shared ownership is used to conceal the larger problem of unclear resource ownership. But there are other downsides to shared ownership. The performance of a shared pointer is always going to be lower than that of a raw pointer. On the other hand, a unique pointer can be just as efficient as a raw pointer (and in fact, `std::unique_ptr` is). When the shared pointer is first created, an additional memory allocation for the reference count must take place. In C++11, `std::make_shared` can be used to combine the allocations for the object itself and the reference counter, but it implies that the object is created with the intent to share (often, the object factory returns unique pointers, some of which are later converted to shared pointers). Copying or deleting a shared pointer must also increment or decrement the reference counter. Shared pointers are often attractive in concurrent data structures, where, at least at the low level, the notion of ownership may indeed be fuzzy, with several accesses to the same object happening at the same time. But designing a shared pointer to be thread-safe in all contexts is not easy and carries additional run-time overhead.

Summary

In C++, memory ownership is really a shorthand for object ownership, which, in turn, is the way to manage arbitrary resources, their ownership, and access. We have reviewed the contemporary idioms the C++ community have developed to express different types of memory ownership. C++ allows the programmer to express exclusive or shared memory ownership. Just as important is expressing *non-ownership* in programs that are agnostic about the ownership of the resources. We have also learned the practices and attributes of the resource ownership in a well-designed program.

Questions

1. Why is it important to clearly express memory ownership in a program?
2. What are the common problems arising from unclear memory ownership?
3. What types of memory ownership can be expressed in C++?
4. How to write non-memory-owning functions and classes?
5. Why should exclusive memory ownership be preferred to the shared one?
6. How to express exclusive memory ownership in C++?
7. How to express shared memory ownership in C++?
8. What are the potential downsides of shared memory ownership?

Further Reading

- *C++: From Beginner to Expert* [Video] by Arkadiusz Wlodarczyk:

  https://www.packtpub.com/application-development/c-beginner-expert-video

- *C++ Data Structures and Algorithms* by Wisnu Anggoro:

  https://www.packtpub.com/application-development/c-data-structures-and-algorithms

- *Expert C++ Programming* by Jeganathan Swaminathan, Maya Posch, Jacek Galowicz:

  https://www.packtpub.com/application-development/expert-c-programming

Template Method Pattern and Non-Virtual Idiom

Coming soon...

Curiously Recurring Template Pattern

Coming soon...

Mixin

Coming soon...

Scopeguard

Coming soon...

Serialization/Deserialization and Virtual Template

Coming soon...

Policy-Based Design

Coming soon...

Policy-Based Design, Inside-Out

Coming soon...

Typelist and Typemap

Coming soon...

Visitor and Multiple Dispatch

Coming soon...