9
A Hands-On Approach to Allocators

In `Chapter 7`, *A Comprehensive Look at Memory Management*, we learned how to allocate and deallocate memory using C++-specific techniques, including the use of `std::unique_ptr` and `std::shared_ptr`. In addition, we learned about fragmentation and how it is capable of wasting large amounts of memory depending on how memory is allocated and then later deallocated. System programmers often have to allocate memory from different pools (sometimes originating from different sources), and handle fragmentation to prevent the system from running out of memory during operation. This is especially true for embedded programmers. Placement `new()` may be used to solve these types of issues, but implementations based on placement new are often hard to create and even harder to maintain. Placement `new()` is also only accessible from user-defined code, providing no control over the allocations that originate from the C++ standard library APIs (such as `std:: list` and `std:: map`).

To solve these types of issues, C++ provides a concept called the **allocator**. C++ allocators define how memory should be allocated and deallocated for a specific type T. In this chapter, you will learn how to create your own allocators while covering the intricate details of the C++ allocator concept. This chapter will end with two different examples; the first example will demonstrate how to create a simple, cache-aligned allocator that is stateless, while the second will provide a functional example of a stateful object allocator that maintains a free pool for fast allocations.

The objectives of this chapter are as follows:

- Introducing the C++ allocators
- Studying an examples of stateless, cache-aligned allocator
- Studying an example of stateful, memory-pool allocator

Technical requirements

In order to compile and execute the examples in this chapter, the reader must have the following:

- A Linux-based system capable of compiling and executing C++17 (for example, Ubuntu 17.10+)
- GCC 7+
- CMake 3.6+
- An internet connection

To download all of the code in this chapter, including the examples and code snippets, please see the following link: `https://github.com/PacktPublishing/Hands-On-System-Programming-with-CPP/tree/master/Chapter09`.

Introducing the C++ allocators

C++ allocators define a template class that allocates memory for a specific type T and are defined by the allocator concept definition. There are two different types of allocators:

- Allocators that are equal
- Allocators that are unequal

An allocator that is equal is an allocator that can allocate memory from one allocator and deallocate memory from another, for example:

```
myallocator<myclass> myalloc1;
myallocator<myclass> myalloc2;

auto ptr = myalloc1.allocate(1);
myalloc2.deallocate(ptr, 1);
```

As shown in the preceding example, we create two instances of `myallocator{}`. We allocate memory from one of the allocators and then deallocate memory from the other allocator. For this to be valid, the allocators must be equal:

```
myalloc1 == myalloc2; // true
```

If this does not hold true, the allocators are considered unequal, which greatly complicates how the allocators can be used. An unequal allocator is usually an allocator that is stateful, meaning it stores a state within itself that prevents an allocator from deallocating memory from another instance of the same allocator (because the state is different).

Learning about the basic allocator

Before we dive into the details of a stateful, unequal allocator, let's review the most basic allocator, which is a stateless, equal allocator. This most basic allocator takes the following form:

```
template<typename T>
class myallocator
{
public:

 using value_type = T;
 using pointer = T *;
 using size_type = std::size_t;

public:

 myallocator() = default;

 template <typename U>
 myallocator(const myallocator<U> &other) noexcept
 { (void) other; }

 pointer allocate(size_type n)
 {
 if (auto ptr = static_cast<pointer>(malloc(sizeof(T) * n))) {
 return ptr;
 }

 throw std::bad_alloc();
 }

 void deallocate(pointer p, size_type n)
 { (void) n; return free(p); }
};

template <typename T1, typename T2>
bool operator==(const myallocator<T1> &, const myallocator<T2> &)
{ return true; }

template <typename T1, typename T2>
bool operator!=(const myallocator<T1> &, const myallocator<T2> &)
{ return false; }
```

To start, all allocators are template classes, as follows:

```
template<typename T>
class myallocator
```

It should be noted that allocators can have any number of template arguments, but at least one is needed to define the type that the allocator will allocate and deallocate. In our example, we use the following aliases:

```
using value_type = T;
using pointer = T *;
using size_type = std::size_t;
```

Technically speaking, the only alias that is required is the following:

```
using value_type = T;
```

Since, however, `T*` and `std::size_t` are required to create a minimal allocator, these aliases might as well be added to provide a more complete implementation. The optional aliases include the following:

```
using value_type = T;
using pointer = T *;
using const_pointer = const T *;
using void_pointer = void *;
using const_void_pointer = const void *;
using size_type = std::size_t;
using difference_type = std::ptrdiff_t;
```

If a custom allocator doesn't provide these, the preceding default values will be provided for you.

As shown, all allocators must provide a default constructor. This is due to the fact that C++ containers will create the allocator on their own, in some cases more than once, and they will use the default constructor to do so, which means the construction of an allocator must be possible without the need of an additional argument.

The `allocate()` function in our example is the following:

```
pointer allocate(size_type n)
{
    if (auto ptr = static_cast<pointer>(malloc(sizeof(T) * n))) {
        return ptr;
    }

    throw std::bad_alloc();
}
```

As with all of the functions being explained in this example, the function signature of the `allocate()` function is defined by the allocator concept, which means that each function in the allocator must take on a specific signature; otherwise, the allocator will not compile correctly when used by existing containers.

In the preceding example, `malloc()` is used to allocate some memory, and if `malloc` doesn't return `nullptr`, the resulting pointer is returned. Since the allocator allocates pointers of the `T*` type, and not `void *`, we must statically cast the result of `malloc()` before returning the pointer. The number of bytes provided to `malloc()` is equal to `sizeof(T) * n`. This is because the `n` parameter defines the total number of objects the allocator must allocate—because some containers will allocate several objects at once and expect that the objects being allocated are contiguous in memory. Examples of this include `std::deque` and `std::vector`, and it's up to the allocator to ensure these rules hold true in memory. Finally, if `malloc()` returns `nullptr`, indicating the requested memory could not be allocated, we throw `std::bad_alloc()`.

It should be noted that in our example, we use `malloc()` instead of `new()`. Here, `malloc()` should be used instead of `new()` because the container will construct the object being allocated for you. For this reason, we don't want to use `new()`, since it would also construct the object, meaning the object would be constructed twice, which would lead to corruption and undefined behavior. For this reason, `new()` and `delete()` should never be used in an allocator.

The `deallocate` function performs the opposite of the `allocate` function, freeing memory and releasing it back to the operating system:

```
void deallocate(pointer p, size_type n)
{ (void) n; free(p); }
```

In the preceding example, to deallocate memory, we simply need to call `free()`. Note that we are creating an *equal* allocator, which means that `ptr` does not need to originate from the same allocator performing the deallocation. The number of allocations, `n`, however, must match the original allocation, which in our case may be safely ignored, since we are using `malloc()` and `free()`, which automatically keep track of the size of the original allocation for us. Not all allocators will have this property.

In our simple example, there are two additional requirements to conform to a C++ allocator that are far less obvious in terms of what exactly their purpose is. The first is the use of a copy constructor using a template type of `U`, as follows:

```
template <typename U>
myallocator(const myallocator<U> &other) noexcept
{ (void) other; }
```

This is because when you use the allocator with a container, you specify the type in the container's definition, for example:

```
std::list<myclass, myallocator<myclass>> mylist;
```

In the preceding example, we create an `std::list` of the `myclass{}` type, with an allocator that allocates and deallocates `myclass{}` objects. The problem is, `std::list` has its own internal data structures that must also be allocated. Specifically, `std::list` implements a linked list, and as a result, `std::list` must be able to allocate and deallocate linked list nodes. In the preceding definition, we defined an allocator that allocates and deallocates `myclass{}` objects, but `std::list` will actually allocate and deallocate nodes and these two types are not the same. To solve this, `std::list` will create a copy of the `myclass{}` allocator using the template version of the copy constructor, providing `std::list` with the ability to create its own node allocator using the allocator that it was originally provided. For this reason, the template version of the copy constructor is required for a fully functional allocator.

The second odd addition to the preceding example is the use of the equality operators, as follows:

```
template <typename T1, typename T2>
bool operator==(const myallocator<T1> &, const myallocator<T2> &)
{ return true; }

template <typename T1, typename T2>
bool operator!=(const myallocator<T1> &, const myallocator<T2> &)
{ return false; }
```

The equality operators define whether the allocator is *equal* or *unequal*. In the preceding example, we have created a stateless allocator, which means that the following is valid:

```
myallocator<int> myalloc1;
myallocator<int> myalloc2;

auto ptr = myalloc1.allocate(1);
myalloc2.deallocate(ptr, 1);
```

If the preceding property holds true, the allocators are equal. Since, in our example, `myalloc1{}` calls `malloc()` when allocating, and `myalloc2{}` calls `free()` when deallocating, we know that they are interchangeable, which means the preceding holds true and our example implements an *equal* allocator. The preceding equality operators simply state this equality formally, providing APIs, such as C++ containers, with a means to create new allocators as needed.

Understanding the allocator's properties and options

The basic allocator we just discussed provides only the required functionality to create and use an allocator with existing C++ data structures (and other user-defined types that leverage object allocation). In addition to the optional aliases we discussed, there are several other options and properties that make up C++ allocators.

Learning the properties

C++ allocators must adhere to a certain set of properties, most of which are either obvious or easily adhered to.

The value pointer type

The first set of properties ensures that the pointer type returned by the allocator is, in fact, a pointer:

```
myallocator<myclass> myalloc;

myclass *ptr = myalloc.allocate(1);
const myclass *cptr = myalloc.allocate(1);

std::cout << (*ptr).data1 << '\n';
std::cout << (*cptr).data2 << '\n';

std::cout << ptr->data1 << '\n';
std::cout << cptr->data2 << '\n';

// 0
// 32644
// 0
// 32644
```

If the pointer returned by the allocator is truly a pointer, it's possible to dereference the pointer to access the memory it points to, as shown in the preceding example. It should also be noted that in this example, we get relatively random values returned when attempting to output the resulting allocated memory to `stdout`. This is because there is no requirement to zero memory from an allocator, as this operation is done for us by the container that uses this memory, which is more performant.

Equality

As stated previously, if an allocator is equal when compared, they return `true`, as shown here:

```
myallocator<myclass> myalloc1;
myallocator<myclass> myalloc2;

std::cout << std::boolalpha;
std::cout << (myalloc1 == myalloc2) << '\n';
std::cout << (myalloc1 != myalloc2) << '\n';

// true
// false
```

If two allocators of the same type return `true`, it means a container that uses this allocator is free to allocate and deallocate memory with different instances of the same allocator freely, which ultimately enables the use of certain optimizations. For example, it's possible for a container to never actually store an internal reference to an allocator, and instead to create an allocator only when memory needs to be allocated. From that point on, the container manages memory internally, and only deallocates memory on destruction, at which time the container will create yet another allocator to perform deallocations, once again assuming the allocators are equal.

As we've covered, allocator equality usually correlates with statefulness. Typically, stateful allocators are not equal, while stateless allocators are equal; but this rule doesn't always hold true, especially when a copy is made of a stateful allocator, which is required by the spec to provide equality (or at least the ability to deallocate previously-allocated memory that was allocated from the copy). We will provide more details on this specific issue when we cover stateful allocators.

One issue with allocators prior to C++17 was that there was no easy way for a container to identify whether an allocator was equal, without first creating two instances of the same allocator at initialization, comparing them, and then setting the internal state based on the result. Due to this limitation in the C++ allocator concept, containers either assumed stateless allocators (which was the case with older versions of C++ libraries), or they assumed all allocators were stateful, removing the possibility of optimizations.

To overcome this, C++17 introduced the following:

```
using is_always_equal = std::true_type;
```

If this is not provided by your allocator, as is the case with the preceding examples, the default value is `std::empty`, telling the container that the old-style comparisons are required to determine equality. If this alias is provided, the container will know how to optimize itself.

Different allocation types

How memory is allocated by a container depends entirely on the type of container, and as a result, an allocator must be able to support different allocation types, such as the following:

- All allocations by an allocator must be contiguous in memory. There is no requirement for one allocation to be contiguous in memory with another allocation, but each individual allocation must be contiguous.
- An allocator must be able to allocate more than one element in a single allocation. This can sometimes be problematic depending on the allocator.

To explore these properties, let's use the following example:

```
template<typename T>
class myallocator
{
public:

    using value_type = T;
    using pointer = T *;
    using size_type = std::size_t;
    using is_always_equal = std::true_type;

public:

    myallocator()
    {
        std::cout << this << " constructor, sizeof(T): "
                  << sizeof(T) << '\n';
    }

    template <typename U>
    myallocator(const myallocator<U> &other) noexcept
    { (void) other; }

    pointer allocate(size_type n)
    {
        if (auto ptr = static_cast<pointer>(malloc(sizeof(T) * n))) {
            std::cout << this << " A [" << n << "]: " << ptr << '\n';
            return ptr;
        }

        throw std::bad_alloc();
    }

    void deallocate(pointer p, size_type n)
    {
        (void) n;

        std::cout << this << " D [" << n << "]: " << p << '\n';
        free(p);
    }
};

template <typename T1, typename T2>
bool operator==(const myallocator<T1> &, const myallocator<T2> &)
{ return true; }

template <typename T1, typename T2>
bool operator!=(const myallocator<T1> &, const myallocator<T2> &)
{ return false; }
```

The preceding allocator is the same as the first allocator, with the exception that debugging statements were added to the constructors and the allocate and deallocate functions, allowing us to see how a container is allocating memory.

Let's examine a simple example of `std::list`:

```
std::list<int, myallocator<int>> mylist;
mylist.emplace_back(42);

// 0x7ffe97b0e8e0 constructor, sizeof(T): 24
// 0x7ffe97b0e8e0 A [1]: 0x55c0793e8580
// 0x7ffe97b0e8e0 D [1]: 0x55c0793e8580
```

As we can see, we have a single allocation and deallocation from the allocator. The allocator is allocating memory of 24 bytes even though the type provided was an int, which is of 4 bytes in size. This is because `std::list` allocates linked list nodes, which in this case are 24 bytes. The allocator is located at `0x7ffe97b0e8e0`, and the allocation was located at `0x55c0793e8580`. Also, as shown, the number of elements allocated each time the allocate function was called was one. This is because `std::list` implements a linked list, which does a dynamic allocation for each element added to the list. Although this seems extremely wasteful when a custom allocator is leveraged, this can be quite helpful when performing system programming as it is sometimes easier to work with memory when only one element is being allocated at a time (instead of multiple).

Now let's look at `std::vector`, as follows:

```
std::vector<int, myallocator<int>> myvector;
myvector.emplace_back(42);
myvector.emplace_back(42);
myvector.emplace_back(42);

// 0x7ffe1db8e2d0 constructor, sizeof(T): 4
// 0x7ffe1db8e2d0 A [1]: 0x55bf9dbdd550
// 0x7ffe1db8e2d0 A [2]: 0x55bf9dbebe90
// 0x7ffe1db8e2d0 D [1]: 0x55bf9dbdd550
// 0x7ffe1db8e2d0 A [4]: 0x55bf9dbdd550
// 0x7ffe1db8e2d0 D [2]: 0x55bf9dbebe90
// 0x7ffe1db8e2d0 D [4]: 0x55bf9dbdd550
```

In the preceding example, we create `std::vector` with our customer allocator, and then, unlike the previous example, we add three integers to the vector instead of one. This is because `std::vector` has to maintain contiguous memory regardless of the number of elements in the vector (which is one of the main properties of `std::vector`). As a result, if `std::vector` fills up (that is, runs out of memory), `std::vector` must allocate a completely new, contiguous block of memory for all of the elements in `std::vector`, copy `std::vector` from the old memory to the new memory, and then deallocate the previous block of memory as it is no longer large enough.

To demonstrate how this works, we add three elements to `std::vector`:

- The first element allocates a block of memory that is four bytes in size (`n == 1` and `sizeof(T) == 4`).
- The second time we add data to `std::vector`, the current block of memory is full (as only four bytes were allocated the first time around), so `std::vector` must deallocate this previously-allocated memory, allocate a new block of memory, and then copy the old contents of `std::vector`. This time around, however, the allocation sets `n == 2`, so eight bytes are allocated.
- The third time we add an element, `std::vector` is out of memory again, and the process is repeated but with `n == 4`, which means that 16 bytes are allocated.

As a side note, the first allocation starts at `0x55bf9dbdd550`, which also happens to be the location of the third allocation. This is because `malloc()` is allocating memory that is aligned to 16 bytes, which means that the first allocation, although only 4 bytes in size, actually allocated 16 bytes, which would have been enough for `n == 4` in the first place (that is, the implementation of `std::vector` provided by GCC could use an optimization). Since the first allocation is deallocated the second time memory is added to the `std::vector`, this memory is free to be used for the third time an element is used, as the original allocation is still large enough for the requested amount.

It is obvious looking at how the allocator is used, that unless you actually need contiguous memory, `std::vector` is not a good choice for storing a list, as it is slow. `std::list`, however, takes up a lot of additional memory, as each element is 24 bytes, instead of 4. The next and final container to observe is `std::deque`, which finds a happy medium between `std::vector` and `std::list`:

```
std::deque<int, myallocator<int>> mydeque;
mydeque.emplace_back(42);
mydeque.emplace_back(42);
mydeque.emplace_back(42);

// constructor, sizeof(T): 4
// 0x7ffdea986e67 A [8]: 0x55d6822b0da0
// 0x7ffdea986f30 A [128]: 0x55d6822afaf0
// 0x7ffdea986f30 D [128]: 0x55d6822afaf0
// 0x7ffdea986e67 D [8]: 0x55d6822b0da0
```

`std::deque` creates a linked list of memory blocks that can be used to store more than one element. In other words, `std::deque` is a `std::list` of `std::vectors`. Like `std::list`, memory is not contiguous, but like `std::vector`, each element only consumes four bytes and a dynamic memory allocation is not needed for each element added. As shown, `sizeof(T) == 4` bytes, and during the creation of `std::deque`, a large buffer of memory is allocated to store several elements (`128` elements, to be specific). The second, smaller allocation is used for internal bookkeeping.

To further explore `std::deque`, let's add a lot of elements to `std::deque`:

```
std::deque<int, myallocator<int>> mydeque;

for (auto i = 0; i < 127; i++)
    mydeque.emplace_back(42);

for (auto i = 0; i < 127; i++)
    mydeque.emplace_back(42);

for (auto i = 0; i < 127; i++)
    mydeque.emplace_back(42);

// constructor, sizeof(T): 4
// 0x7ffc5926b1b7 A [8]: 0x560285cc0da0
// 0x7ffc5926b280 A [128]: 0x560285cbfaf0
// 0x7ffc5926b280 A [128]: 0x560285cc1660
// 0x7ffc5926b280 A [128]: 0x560285cc1bc0
// 0x7ffc5926b280 D [128]: 0x560285cbfaf0
// 0x7ffc5926b280 D [128]: 0x560285cc1660
// 0x7ffc5926b280 D [128]: 0x560285cc1bc0
// 0x7ffc5926b1b7 D [8]: 0x560285cc0da0
```

In the preceding example, we add `127` elements three times. This is because each allocation allocates enough for `128` elements, with one of the elements being used for bookkeeping. As shown, `std::deque` allocates three blocks of memory.

Copying equal allocators

Copying containers with allocators that are equal is straightforward—this is because the allocators are interchangeable. To explore this, let's add the following overloads to the previous allocator so that we may observe additional operations taking place:

```
myallocator(myallocator &&other) noexcept
{
    (void) other;
    std::cout << this << " move constructor, sizeof(T): "
```

```
                 << sizeof(T) << '\n';
}

myallocator &operator=(myallocator &&other) noexcept
{
    (void) other;
    std::cout << this << " move assignment, sizeof(T): "
                 << sizeof(T) << '\n';
    return *this;
}

myallocator(const myallocator &other) noexcept
{
    (void) other;
    std::cout << this << " copy constructor, sizeof(T): "
                 << sizeof(T) << '\n';
}

myallocator &operator=(const myallocator &other) noexcept
{
    (void) other;
    std::cout << this << " copy assignment, sizeof(T): "
                 << sizeof(T) << '\n';
    return *this;
}
```

The preceding code adds a copy constructor, `copy assignment` operator, move constructor, and a `move assignment` operator, all of which have debug statements so that we may see what the container is doing. With the preceding addition, we will be able to see when a copy of an allocator is performed. Now let's use this allocator in a container that is copied:

```
std::list<int, myallocator<int>> mylist1;
std::list<int, myallocator<int>> mylist2;

mylist1.emplace_back(42);
mylist1.emplace_back(42);

std::cout << "----------------------------------------\n";
mylist2 = mylist1;
std::cout << "----------------------------------------\n";

mylist2.emplace_back(42);
mylist2.emplace_back(42);
```

In the preceding example, we create two lists. In the first `std::list`, we add two elements to the list and then we copy the list to the second `std::list`. Finally, we add two more elements to the second `std::list`. The output is as follows:

```
0x7fff866d1e50 constructor, sizeof(T): 24
0x7fff866d1e70 constructor, sizeof(T): 24
0x7fff866d1e50 A [1]: 0x557c430ec550
0x7fff866d1e50 A [1]: 0x557c430fae90
----------------------------------------
0x7fff866d1d40 copy constructor, sizeof(T): 24
0x7fff866d1d40 A [1]: 0x557c430e39a0
0x7fff866d1d40 A [1]: 0x557c430f14a0
----------------------------------------
0x7fff866d1e70 A [1]: 0x557c430f3b30
0x7fff866d1e70 A [1]: 0x557c430ec4d0
0x7fff866d1e70 D [1]: 0x557c430e39a0
0x7fff866d1e70 D [1]: 0x557c430f14a0
0x7fff866d1e70 D [1]: 0x557c430f3b30
0x7fff866d1e70 D [1]: 0x557c430ec4d0
0x7fff866d1e50 D [1]: 0x557c430ec550
0x7fff866d1e50 D [1]: 0x557c430fae90
```

As expected, each list creates the allocator that it plans to use, and the allocators create `std::list` nodes of 24 bytes. We then see the first allocator allocate memory for the two elements that are added to the first list. The second list is still empty just prior to copying the first list and, as a result, the second container creates a third, temporary allocator that it can use solely for copying the lists. Once this is done, we add the final two elements to the second list, and we can see the second list uses its original allocator to perform the allocations.

`std::list` is free to allocate memory from one allocator and deallocate from another, and this is seen in the deallocations, which is why `std::list` creates a temporary allocator during the copy, as it is free to do so. Whether a container should create temporary allocators is not the point (although it is likely a debatable optimization).

Moving equal allocators

Moving a container is similar to copying a container if the allocators are equal. Once again, this is because there are no rules as to what the container has to do, since a container can use its original allocator to handle any memory, and if it needs to, it can create a new allocator, as follows:

```
std::list<int, myallocator<int>> mylist1;
std::list<int, myallocator<int>> mylist2;
```

```
    mylist1.emplace_back(42);
    mylist1.emplace_back(42);

    std::cout << "----------------------------------------\n";
    mylist2 = std::move(mylist1);
    std::cout << "----------------------------------------\n";

    mylist2.emplace_back(42);
    mylist2.emplace_back(42);
```

In the preceding example, instead of copying the first container, we move it instead. As a result, the first container after the move is no longer valid, and the second container now owns the memory from the first container.

The output of this example is as follows:

```
0x7ffe582e2850 constructor, sizeof(T): 24
0x7ffe582e2870 constructor, sizeof(T): 24
0x7ffe582e2850 A [1]: 0x56229562d550
0x7ffe582e2850 A [1]: 0x56229563be90
----------------------------------------
----------------------------------------
0x7ffe582e2870 A [1]: 0x5622956249a0
0x7ffe582e2870 A [1]: 0x5622956324a0
0x7ffe582e2870 D [1]: 0x56229562d550
0x7ffe582e2870 D [1]: 0x56229563be90
0x7ffe582e2870 D [1]: 0x5622956249a0
0x7ffe582e2870 D [1]: 0x5622956324a0
```

Similar to the copy example, the two lists are created and each `std::list` creates an allocator that manages the `std::list` nodes of 24 bytes. Two elements are added to the first list, and then the first list is moved into the second list. As a result, memory that belongs to the first list is now owned by the second container and no copies are performed. The second allocations to the second list are performed by its own allocator, as are all deallocations, since allocations from the first allocator can be deallocated using the second allocator.

Exploring some optional properties

C++ allocators provide some additional properties that are above and beyond `is_always_equal`. Specifically, the author of a C++ allocator can optionally define the following:

- `propagate_on_container_copy_assignment`
- `propagate_on_container_move_assignment`
- `propagate_on_container_swap`

The optional properties tell a container how the allocator should be handled during a specific operation (that is, copy, move, and swap). Specifically, when a container is copied, moved, or swapped, the allocator isn't touched and, as we will show, this can result in inefficiencies. The propagate properties tell the container to propagate the operation to the allocator. For example, if `propagate_on_container_copy_assignment` is set to `std::true_type` and a container is being copied, the allocator must also be copied when normally it wouldn't be.

To better explore these properties, let's create our first unequal allocator (that is, two different instances of the same allocator may not be equal). As stated, most allocators that are unequal are stateful. In this example, we will create a stateless, unequal allocator to keep things simple. Our last example in this chapter will create an unequal, stateful allocator.

To start our example, we first need to create a managed object for our allocator class, as follows:

```
class myallocator_object
{
public:

    using size_type = std::size_t;

public:

    void *allocate(size_type size)
    {
        if (auto ptr = malloc(size)) {
            std::cout << this << " A " << ptr << '\n';
            return ptr;
        }

        throw std::bad_alloc();
    }
```

```
        void deallocate(void *ptr)
        {
            std::cout << this << " D " << ptr << '\n';
            free(ptr);
        }
    };
```

Unequal allocators must adhere to the following properties:

- All copies of an allocator must be equal. This means that even if we create an unequal allocator, a copy of an allocator must still be equal. This becomes problematic when the rebind copy constructor is used, as this property still holds true (that is, even though two allocators may not have the same type, they may still have to be equal if one is the copy of another).
- All equal allocators must be able to deallocate each other's memory. Once again, this becomes problematic when the rebind copy constructor is used. Specifically, this means that an allocator managing `int` objects might have to deallocate memory from an allocator managing `std::list` nodes.

To support these two rules, most unequal allocators end up being wrappers around a managed object. That is, an object is created that can allocate and deallocate memory and each allocator stores a pointer to this object. In the preceding example, `myallocator_object{}` is the managed object capable of allocating and deallocating memory. To create this object, all we did was move `malloc()` and `free()` from the allocator itself into this `myallocator_object{}`; the code is the same. The only additional logic that was added to `myallocator_object{}` is the following:

- The constructor takes a size. This is because we cannot create the managed object as a template class. Specifically, the managed object needs to be able to change the type of memory that it manages (because of the rules outlined). The specific need for this will be covered shortly.
- A `rebind()` function was added that specifically changes the size of the memory being managed by the managed object. Once again, this allows us to change the size of the allocation being performed by `myallocator_object{}`.

Next, we need to define the allocator itself, as follows:

```
    template<typename T>
    class myallocator
    {
```

The first part of the allocator is the same as the other allocators, requiring the use of a template class that allocators memory for some `T` type:

```
public:

    using value_type = T;
    using pointer = T *;
    using size_type = std::size_t;
    using is_always_equal = std::false_type;
```

The next part of our allocator defines our type aliases and optional properties. As shown, all three propagate functions are undefined, which tells any container that uses this allocator that when a copy, move, or swap of the container occurs, the allocator is not copied, moved, or swapped as well (the container should continue using the same allocator it was given at construction).

The next set of functions defines our constructors and operators. Let's start with the default constructor:

```
myallocator() :
    m_object{std::make_shared<myallocator_object>()}
{
    std::cout << this << " constructor, sizeof(T): "
                << sizeof(T) << '\n';
}
```

As with all of the constructors and operators, we output to `stdout` some debug information so that we can watch what the container is doing with the allocator. As shown, the default constructor allocates `myallocator_object{}` and stores it as `std::shared_ptr`. We leverage `std::shared_ptr`, as each copy of the allocator will have to be equal, and as a result, each copy must share the same managed object (so that memory allocated from one allocator can be deallocated from the copy). Since either allocator could be destroyed at any type, both *own* the managed object and as a result, `std::shared_ptr` is the more appropriate smart pointer.

The next two functions are the move constructor and assignment operator:

```
myallocator(myallocator &&other) noexcept :
    m_object{std::move(other.m_object)}
{
    std::cout << this << " move constructor, sizeof(T): "
                << sizeof(T) << '\n';
}

myallocator &operator=(myallocator &&other) noexcept
{
    std::cout << this << " move assignment, sizeof(T): "
                << sizeof(T) << '\n';

    m_object = std::move(other.m_object);
    return *this;
}
```

In both cases, we need to `std::move()` our managed object as a result of a move operation. The same thing applies for copying as well:

```
    myallocator(const myallocator &other) noexcept :
        m_object{other.m_object}
    {
        std::cout << this << " copy constructor, sizeof(T): "
                  << sizeof(T) << '\n';
    }

    myallocator &operator=(const myallocator &other) noexcept
    {
        std::cout << this << " copy assignment, sizeof(T): "
                  << sizeof(T) << '\n';

        m_object = other.m_object;
        return *this;
    }
```

As shown, if a copy is made of the allocator, we must also copy the managed object. As a result, a copy of the allocator leverages the same managed object, which means that the copy can deallocate memory from the original.

The next function is what makes unequal allocators so difficult:

```
    template <typename U>
    myallocator(const myallocator<U> &other) noexcept :
        m_object{other.m_object}
    {
        std::cout << this << " copy constructor (U), sizeof(T): "
                  << sizeof(T) << '\n';
    }
```

The preceding function is the rebind copy constructor. The point of this constructor is to create a copy of another allocator of a different type. So for example, `std::list` starts off with `myallocator<int>{}`, but it really needs an allocator of the `myallocator<std::list::node>{}` type, not `myallocator<int>{}`. To overcome this, the preceding function allows a container to do something like the following:

```
myallocator<int> alloc1;
myallocator<std::list::node> alloc2(alloc1);
```

In the preceding example, `alloc2` is a copy of `alloc1`, even though `alloc1` and `alloc2` do not share the same `T` type. The problem is, an `int` is four bytes, while in our examples, `std::list::node` has been 24 bytes, which means that not only does the preceding function have to be able to create a copy of an allocator with a different type that is *equal*, it also has to be able to create a copy that is capable of deallocating memory of a different type (specifically, in this case, `alloc2` has to be able to deallocate ints even though it manages `std::list::node` elements). In our example, this is not a problem since we are using `malloc()` and `free()`, but as we will show in our last example, some stateful allocators, such as a memory pool, do not conform well to this requirement.

The `allocate` and `deallocate` functions are defined as follows:

```
pointer allocate(size_type n)
{
    auto ptr = m_object->allocate(sizeof(T) * n);
    return static_cast<pointer>(ptr);
}

void deallocate(pointer p, size_type n)
{
    (void) n;
    return m_object->deallocate(p);
}
```

Since our managed object just calls `malloc()` and `free()`, we can treat the object's `allocate()` and `deallocate()` functions as `malloc()` and `free()` as well, and as such, the implementation is simple.

Our private logic in the `allocator` class is as follows:

```
std::shared_ptr<myallocator_object> m_object;

template <typename T1, typename T2>
friend bool operator==(const myallocator<T1> &lhs, const myallocator<T2>
&rhs);
```

```
template <typename T1, typename T2>
friend bool operator!=(const myallocator<T1> &lhs, const myallocator<T2>
&rhs);
```

As stated, we store a smart pointer to the managed object, which allows us to create copies of the allocator. We also state that our equality functions are friends, and although we place these friend functions in the private portion of the class, we could have placed them anywhere as friend declarations are not affected by public/protected/private declarations.

Finally, the equality functions are as follows:

```
template <typename T1, typename T2>
bool operator==(const myallocator<T1> &lhs, const myallocator<T2> &rhs)
{ return lhs.m_object.get() == rhs.m_object.get(); }

template <typename T1, typename T2>
bool operator!=(const myallocator<T1> &lhs, const myallocator<T2> &rhs)
{ return lhs.m_object.get() != rhs.m_object.get(); }
```

Our *equal* allocator example simply returned true for `operator==` and false for `operator!=`, which stated that the allocators were equal (in addition to the use of `is_always_equal`). In this example, `is_always_equal` is set to `false`, and in our equality operators, we compare the managed objects. Each time a new allocator is created, a new managed object is created, and as a result, the allocators are not equal (that is, they are unequal allocators). The problem is, we cannot simply always return `false` for `operator==` because a copy of an allocator must always be equal to the original per the specification, which is the reason we use `std::shared_ptr`. Each copy of the allocator creates a copy of `std::shared_ptr`, and since we compare the address of the managed object if a copy of the allocator is made, the copy and the original have the same managed object and as a result, return `true` (that is, they are equal). Although `std::shared_ptr` may not be used, most unequal allocators are implemented this way, as it provides a simple way to handle the difference between equal and unequal allocators based on whether or not the allocator has been copied.

Now that we have an allocator, let's test it:

```
std::list<int, myallocator<int>> mylist;
mylist.emplace_back(42);

// 0x7ffce60fbd10 constructor, sizeof(T): 24
// 0x561feb431590 A [1]: 0x561feb43fec0
// 0x561feb431590 D [1]: 0x561feb43fec0
```

As you can see, our allocator is capable of allocating and deallocating memory. The allocator in the preceding example was located at `0x561feb431590`, and the element that was allocated by the `std::list` container was located at `0x561feb43fec0`.

Copying an unequal container that has the propagate property set to `false` is simple, as follows:

```
std::list<int, myallocator<int>> mylist1;
std::list<int, myallocator<int>> mylist2;

mylist1.emplace_back(42);
mylist1.emplace_back(42);

mylist2.emplace_back(42);
mylist2.emplace_back(42);

std::cout << "----------------------------------------\n";
mylist2 = mylist1;
std::cout << "----------------------------------------\n";

mylist2.emplace_back(42);
mylist2.emplace_back(42);
```

As shown in the preceding example, we create two lists and populate both lists with two elements each. Once the lists are populated, we then copy the first container into the second, and we output to `stdout` so that we can see how the container handles this copy. Finally, we add two more elements to the just-copied container.

The output of this example is as follows:

```
// 0x7ffd65a15cb0 constructor, sizeof(T): 24
// 0x7ffd65a15ce0 constructor, sizeof(T): 24
// 0x55c4867c3a80 A [1]: 0x55c4867b9210  <--- add to list #1
// 0x55c4867c3a80 A [1]: 0x55c4867baec0  <--- add to list #1
// 0x55c4867d23c0 A [1]: 0x55c4867c89c0  <--- add to list #2
// 0x55c4867d23c0 A [1]: 0x55c4867cb050  <--- add to list #2
// ----------------------------------------
// ----------------------------------------
// 0x55c4867d23c0 A [1]: 0x55c4867c39f0  <--- add to list #2 after copy
// 0x55c4867d23c0 A [1]: 0x55c4867c3a10  <--- add to list #2 after copy
// 0x55c4867d23c0 D [1]: 0x55c4867c89c0  <--- deallocate list #2
// 0x55c4867d23c0 D [1]: 0x55c4867cb050  <--- deallocate list #2
// 0x55c4867d23c0 D [1]: 0x55c4867c39f0  <--- deallocate list #2
// 0x55c4867d23c0 D [1]: 0x55c4867c3a10  <--- deallocate list #2
// 0x55c4867c3a80 D [1]: 0x55c4867b9210  <--- deallocate list #1
// 0x55c4867c3a80 D [1]: 0x55c4867baec0  <--- deallocate list #1
```

As shown, copying the containers does not involve the allocator. When the copy occurs, list two keeps the two allocations it already has, overwriting the values for the first two elements. Since the propagate properties are `false`, the second container keeps the allocator it was originally given, and uses the allocator to allocate the second two elements after the copy, but also deallocate all of the previously-allocated elements when the list loses scope.

The problem with this approach is the need for the container to loop through each element and perform a manual copy. For integers, this type of copy is fine, but we could have stored large structures in the list and as a result, copying the containers would have resulted in copying each element in the container, which is wasteful and expensive. Since the propagate property is `false`, the container has no choice as it cannot use the allocator from the first list and it cannot use its own allocator to copy the elements allocated in the first list (since the allocators are not equal). Although this is wasteful, as will be shown, this approach may still be the fastest approach.

Moving a list has a similar issue:

```
std::list<int, myallocator<int>> mylist1;
std::list<int, myallocator<int>> mylist2;

mylist1.emplace_back(42);
mylist1.emplace_back(42);

mylist2.emplace_back(42);
mylist2.emplace_back(42);

std::cout << "----------------------------------------\n";
mylist2 = std::move(mylist1);
std::cout << "----------------------------------------\n";

mylist2.emplace_back(42);
mylist2.emplace_back(42);
```

In the preceding example, we do the same thing we did in the previous example. We create two lists, and add two elements to each list just before moving one list to another.

The results of this example are as follows:

```
// 0x7ffd65a15cb0 constructor, sizeof(T): 24
// 0x7ffd65a15ce0 constructor, sizeof(T): 24
// 0x55c4867c3a80 A [1]: 0x55c4867c3a10  <--- add to list #1
// 0x55c4867c3a80 A [1]: 0x55c4867c39f0  <--- add to list #1
// 0x55c4867d23c0 A [1]: 0x55c4867c0170  <--- add to list #2
// 0x55c4867d23c0 A [1]: 0x55c4867c0190  <--- add to list #2
// ----------------------------------------
// ---------------------------------------
// 0x55c4867d23c0 A [1]: 0x55c4867b9c90  <--- add to list #2 after move
// 0x55c4867d23c0 A [1]: 0x55c4867b9cb0  <--- add to list #2 after move
// 0x55c4867d23c0 D [1]: 0x55c4867c0170  <--- deallocate list #2
// 0x55c4867d23c0 D [1]: 0x55c4867c0190  <--- deallocate list #2
// 0x55c4867d23c0 D [1]: 0x55c4867b9c90  <--- deallocate list #2
// 0x55c4867d23c0 D [1]: 0x55c4867b9cb0  <--- deallocate list #2
// 0x55c4867c3a80 D [1]: 0x55c4867c3a10  <--- deallocate list #1
// 0x55c4867c3a80 D [1]: 0x55c4867c39f0  <--- deallocate list #1
```

In the preceding example, we can see that the same inefficiency exists. Since the propagate property is `false`, the container cannot use the allocator from the first list and instead, must continue to use the allocator that it already has. As a result, the move operation cannot simply move the internal container from one list to another, but instead it must loop through the entire container, executing `std::move()` on each individual element such that the memory associated with each node in the list is still managed by the second list's original allocator.

To overcome these issues, we will add the following to our allocator:

```
using propagate_on_container_copy_assignment = std::true_type;
using propagate_on_container_move_assignment = std::true_type;
using propagate_on_container_swap = std::true_type;
```

These properties tell any container that uses this allocator that if a copy, move, or swap of the container occurs, the same operation should occur with the allocator. For example, if we copy `std::list`, the container must not only *copy* the elements, but it should also copy the allocator.

Let's look at the following copy example:

```
std::list<int, myallocator<int>> mylist1;
std::list<int, myallocator<int>> mylist2;

mylist1.emplace_back(42);
mylist1.emplace_back(42);

mylist2.emplace_back(42);
mylist2.emplace_back(42);

std::cout << "----------------------------------------\n";
mylist2 = mylist1;
std::cout << "----------------------------------------\n";

mylist2.emplace_back(42);
mylist2.emplace_back(42);
```

This copy example is the same as our previous copy example. We create two lists and add two elements to each list. We then copy the first list into the second list and then add two additional elements into the second list before finishing (which ultimately will deallocate the lists).

The results of this example are as follows. It should be noted that this output is a bit more complicated, so we will take this one step at a time:

```
// 0x7ffc766ec580 constructor, sizeof(T): 24
// 0x7ffc766ec5b0 constructor, sizeof(T): 24
// 0x5638419d9720 A [1]: 0x5638419d0b60  <--- add to list #1
// 0x5638419d9720 A [1]: 0x5638419de660  <--- add to list #1
// 0x5638419e8060 A [1]: 0x5638419e0cf0  <--- add to list #2
// 0x5638419e8060 A [1]: 0x5638419d9690  <--- add to list #2
```

In the preceding output, both lists are created and two elements are added to each container. Next, the output will show what happens when we copy the second container into the first:

```
// 0x5638419e8060 D [1]: 0x5638419e0cf0
// 0x5638419e8060 D [1]: 0x5638419d9690
// 0x7ffc766ec5b0 copy assignment, sizeof(T): 24
// 0x7ffc766ec450 copy constructor (U), sizeof(T): 4
// 0x7ffc766ec3f0 copy constructor (U), sizeof(T): 24
// 0x7ffc766ec460 copy constructor, sizeof(T): 24
// 0x5638419d9720 A [1]: 0x5638419e8050
// 0x5638419d9720 A [1]: 0x5638419d9690
```

Since we set the propagate property to `false`, the container now has the option to keep the memory used by the first container (for example, to implement a copy-on-write implementation). This is because the container should create a copy of the allocator and any two copies of an allocator are equal (that is, they can deallocate each other's memory). This implementation of glibc does not do this. Instead, it attempts to create a clean view of memory. The two lists, allocators are not equal, which means that once the copy has taken place, the container will no longer be able to deallocate its own, previously-allocated memory (because it will likely no longer have access to its original allocator). As a result, the container deletes all of the memory it previously allocated as its first step. It then creates a temporary allocator using a rebind copy of the first list's allocator (which oddly seems to be unused), just before creating a direct copy of the first list's allocator and using it to allocator new memory for the elements that will be copied.

Finally, now that the copy is complete, the last two elements can be added to the second list, and each list can be destroyed once they lose scope:

```
// 0x5638419d9720 A [1]: 0x5638419d96b0  <--- add to list #2 after copy
// 0x5638419d9720 A [1]: 0x5638419d5e10  <--- add to list #2 after copy
// 0x5638419d9720 D [1]: 0x5638419e8050  <--- deallocate list #2
// 0x5638419d9720 D [1]: 0x5638419d9690  <--- deallocate list #2
// 0x5638419d9720 D [1]: 0x5638419d96b0  <--- deallocate list #2
// 0x5638419d9720 D [1]: 0x5638419d5e10  <--- deallocate list #2
// 0x5638419d9720 D [1]: 0x5638419d0b60  <--- deallocate list #1
// 0x5638419d9720 D [1]: 0x5638419de660  <--- deallocate list #1
```

As shown, since the allocator was propagated, the same allocator is used to deallocate the elements from both lists. This is because once the copy is complete, both lists are now using the same allocator (as a copy of any two allocators must be equal, and the way that we chose to implement this was to create a copy of the same base allocator object when a copy occurs). It should also be noted that the glibc implementation does not choose to implement a copy-on-write scheme, which means not only that the implementation fails to take advantage of the possible optimizations that the propagate property provides, but the implementation of a copy is actually slower, as the copy not only has to copy each element one at a time, but must also allocate new memory for the copy as well.

Now let's look at a move example:

```
std::list<int, myallocator<int>> mylist1;
std::list<int, myallocator<int>> mylist2;

mylist1.emplace_back(42);
mylist1.emplace_back(42);

mylist2.emplace_back(42);
mylist2.emplace_back(42);

std::cout << "----------------------------------------\n";
mylist2 = std::move(mylist1);
std::cout << "----------------------------------------\n";

mylist2.emplace_back(42);
mylist2.emplace_back(42);
```

Like our previous move example, this creates two lists, and adds two elements to each list just before moving the first list into the second. Finally, our example adds two elements to the second list (which is now the first list), before completing and deallocating both lists when they lose scope.

The resulting output of this example is as follows:

```
// 0x7ffc766ec580 constructor, sizeof(T): 24
// 0x7ffc766ec5b0 constructor, sizeof(T): 24
// 0x5638419d9720 A [1]: 0x5638419d96b0  <--- add to list #1
// 0x5638419d9720 A [1]: 0x5638419d9690  <--- add to list #1
// 0x5638419d5e20 A [1]: 0x5638419e8050  <--- add to list #2
// 0x5638419d5e20 A [1]: 0x5638419d5e30  <--- add to list #2
// ----------------------------------------
// 0x5638419d5e20 D [1]: 0x5638419e8050  <--- deallocate list #2
// 0x5638419d5e20 D [1]: 0x5638419d5e30  <--- deallocate list #2
// 0x7ffc766ec5b0 move assignment, sizeof(T): 24
// ----------------------------------------
// 0x5638419d9720 A [1]: 0x5638419d5e10
// 0x5638419d9720 A [1]: 0x5638419e8050
// 0x5638419d9720 D [1]: 0x5638419d96b0  <--- deallocate list #1
// 0x5638419d9720 D [1]: 0x5638419d9690  <--- deallocate list #1
// 0x5638419d9720 D [1]: 0x5638419d5e10  <--- deallocate list #2
// 0x5638419d9720 D [1]: 0x5638419e8050  <--- deallocate list #2
```

Like the previous examples, you can see the lists being created and the first elements being added to each list. Once the move occurs, the second list deletes the memory associated with its previously-added elements. This is because once the move occurs, the memory associated with the second list is no longer needed (as it is about to be replaced with the memory allocated by the first list). This is possible since the first list's allocator will be moved to the second list (since the propagate property was set to `true`), and as a result, the second list will now own all of the first list's memory.

Finally, the last two elements are added to the list and the lists lose scope and deallocate all of their memory. As shown, this is the most optimal implementation. No additional memory is allocated no element-by-element move is needed. The move operation simply moves the memory and allocator from one container to the other. Also, since no copy of the allocators is made, this is a simple operation for any allocator to support, and as such, this property should always be set to true.

Optional functions

In addition to properties, there are several optional functions that provide containers with additional information about the type of allocator they are provided. One optional function is the following:

```
size_type myallocator::max_size();
```

The max_size() function tells the container the max size, "n", that an allocator can allocate. In C++17, this function has been deprecated. The max_size() function returns the largest possible allocation that the allocator can perform. Curiously, in C++17, this defaults to std::numeric_limits<size_type>::max() / sizeof(value_type), which in most cases is likely not a valid answer as most systems simply do not have this much available RAM, suggesting this function provides little value in practice. Instead, like other allocation schemes in C++, std::bad_alloc will be thrown if and when an allocation fails, indicating to the container that the allocation it attempted to perform is not possible.

Another set of optional functions in C++ is the following:

```
template<typename T, typename... Args>
static void myallocator::construct(T* ptr, Args&&... args);

template<typename T>
static void myallocator::destroy(T* ptr);
```

Just like with the max_size() function, the construct and destruct functions were deprecated in C++17. Prior to C++17, these functions could be used to construct and destruct the object associated with the provided by ptr. It should be noted that this is why we do not use new and delete when allocating memory in a constructor, but instead use malloc() and free(). If we were to use new() and delete(), we would accidentally call the constructor and/or destructor of the object twice, which would lead to undefined behavior.

Studying an example of stateless, cache–aligned allocator

In this example, we will create a stateless, equal allocator designed to allocator cache-aligned memory. The goal of this allocator is to show a C++17 allocator that can be leveraged to increase the efficiency of the objects a container is storing (for example, a linked list), as cache-thrashing is less likely to occur.

To start, we will define the allocator as follows:

```
template<typename T, std::size_t Alignment = 0x40>
class myallocator
{
public:

    using value_type = T;
    using pointer = T *;
```

```
        using size_type = std::size_t;
        using is_always_equal = std::true_type;

        template<typename U> struct rebind {
            using other = myallocator<U, Alignment>;
        };

    public:

        myallocator()
        { }

        template <typename U>
        myallocator(const myallocator<U, Alignment> &other) noexcept
        { (void) other; }

        pointer allocate(size_type n)
        {
            if (auto ptr = aligned_alloc(Alignment, sizeof(T) * n)) {
                return static_cast<pointer>(ptr);
            }

            throw std::bad_alloc();
        }

        void deallocate(pointer p, size_type n)
        {
            (void) n;
            free(p);
        }
    };
```

The preceding allocator is similar to the other equal allocators that we have created in this chapter. There are a couple of notable differences:

- The template signature of the allocator is different. Instead of just defining the allocator type `T`, we also added an `Alignment` parameter and set the default value to `0x40` (that is, the allocations will be 64-byte-aligned, which is the typical size of a cache line on Intel CPUs).
- We also provide our own rebind structure. Typically, this structure is provided for us, but since our allocator has more than one template argument, we must provide our own version of the rebind structure. This structure is used by a container, such as `std::list`, to create any allocator the container needs without having to create a copy (instead, it can directly create an allocator during initialization). In our version of this rebind structure, we pass the `Alignment` parameter that is provided by the original allocator.

- The rebind copy constructor must also define the `Alignment` variable. In this case, we force the `Alignment` to be the same if a rebind is going to occur, which will be the case as the rebind structure provides the `Alignment` (which is also the same).

To test our example, let's create the allocator and output the address of an allocation to ensure that the memory is aligned:

```
myallocator<int> myalloc;

auto ptr = myalloc.allocate(1);
std::cout << ptr << '\n';
myalloc.deallocate(ptr, 1);

// 0x561d512b6500
```

As shown, the memory that was allocated is at least 64-byte-aligned. The same thing is true for multiple allocations, as follows:

```
myallocator<int> myalloc;

auto ptr = myalloc.allocate(42);
std::cout << ptr << '\n';
myalloc.deallocate(ptr, 42);

// 0x55dcdcb41500
```

As shown, the memory allocated is also at least 64-byte-aligned. We can also use this allocator with a container:

```
std::vector<int, myallocator<int>> myvector;
myvector.emplace_back(42);

std::cout << myvector.data() << '\n';

// 0x55f875a0f500
```

And once again, the memory is still properly aligned.

Compiling and testing

To compile this code, we leverage the same `CMakeLists.txt` file that we have been using for the other examples: `https://github.com/PacktPublishing/Hands-On-System-Programming-with-CPP/blob/master/Chapter09/CMakeLists.txt`.

With this code in place, we can compile this code using the following:

```
> git clone
https://github.com/PacktPublishing/Hands-On-System-Programming-with-CPP.git
> cd Hands-On-System-Programming-with-CPP/Chapter09/
> mkdir build
> cd build

> cmake ..
> make
```

To execute the example, run the following command:

```
> ./example6
```

The output should resemble the following:

```
0x55aec04dbd00
0x55aec04e8f40
0x55aec04d5d00
===========================================================================
====
test cases: 3 | 3 passed
assertions: - none -
```

As shown in the preceding snippet, we are able to allocate different types of memory, as well as deallocate this memory and all of the addresses are 64-byte-aligned.

Studying an example of a stateful, memory–pool allocator

In this example, we will create a far more complicated allocator, called a **pool allocator**. The goal of the pool allocator is to quickly allocate memory for a fixed-size type while simultaneously (and more importantly) reducing internal fragmentation of memory (that is, the amount of memory that is wasted by each allocation, even if the allocation size is not a multiple of two or some other optimized allocation size).

Memory-pool allocators are so useful that some implementations of C++ already contain pool allocators. In addition, C++17 technically has support for a pool allocator in something called a **polymorphic allocator** (which is not covered in this book, as no major implementations of C++17 have support for polymorphic allocators at the time of writing), and most operating systems leverage pool allocators within the kernel to reduce internal fragmentation.

The major advantages of a pool allocator are as follows:

- The use of `malloc()` is slow. Sometimes `free()` is slow too, but for some implementations, `free()` is as simple as flipping a bit, in which case it can be implemented incredibly fast.
- Most pool allocators leverage a deque structure, meaning the pool allocator allocates a large *block* of memory and then divides this memory up for allocations. Each *block* of memory is linked using a linked list so that more memory can be added to the pool as needed.

Pool allocators also have an interesting property where the larger the block size, the larger the reduction is on internal fragmentation. The penalty for this optimization is that if the pool is not completely utilized, the amount of memory that is wasted increases as the block size increases, so pool allocators should be tailored to meet the needs of the application.

To start our example, we will first create a `pool` class that manages a list of *blocks* and gives out memory from the blocks. The list of blocks will be stored in a stack that grows forever (that is, in this example, we will attempt to defragment the memory in the blocks, or remove a block from the stack if all memory from the block has been freed). Each time we add a block of memory to the pool, we will divide up the block into chunks of the size of `sizeof(T)`, and add the address of each chunk onto a second stack called the address stack. When memory is allocated, we will pop an address off the address stack, and when memory is deallocated, we will push the address back onto the stack.

The beginning of our pool is as follows:

```
class pool
{
public:

    using size_type = std::size_t;

public:

    explicit pool(size_type size) :
        m_size{size}
    { }
```

The pool will act as our managed object for our unequal allocator, as was the case with our previous unequal allocator example. As a result, the pool is not a template class, as we will need to change the size of the pool if the rebind copy constructor is used (more on that specific topic to come). As shown, in our constructor, we store the size of the pool, but we do not attempt to preload the pool.

To allocate, we pop an address from our address stack and return it. If the address stack is empty, we add more addresses to the address stack by allocating another block of memory, adding it to the stack of blocks, dividing up the memory into chunks, and adding the divided up chunks to the address stack, as follows:

```
void *allocate()
{
    if (m_addrs.empty())
    {
        this->add_addrs();
    }

    auto ptr = m_addrs.top();
    m_addrs.pop();

    return ptr;
}
```

To deallocate memory, we push the address provided to the address stack so that it can be allocated again later on. Using this method, allocating and deallocating memory for a container is as simple as popping and pushing an address to a single stack:

```
void deallocate(void *ptr)
{
    m_addrs.push(ptr);
}
```

We will need to change the size of the pool if the rebind copy constructor is used. This type of copy should only occur when attempting to create an allocator of the `int` type to an allocator of the `std::list::node` type, which means that the allocator being copied will not have been used yet, meaning a resize is possible. If the allocator has been used, it would mean that the allocator has already allocated memory of a different size and, as a result, a rebind would be impossible with this implementation. Consider the following code for it:

```
void rebind(size_type size)
{
    if (!m_addrs.empty() || !m_blocks.empty())
    {
        std::cerr << "rebind after alloc unsupported\n";
        abort();
    }

    m_size = size;
}
```

It should be noted that there are other ways to handle this specific issue. For example, a std::list could be created that doesn't attempt to use the rebind copy constructor. An allocator could also be created that is capable of managing more than one pool of memory, each pool being capable of allocating and deallocating memory of a specific type (which, of course, would result in a performance hit).

In our private section, we have the add_addrs() function that was seen in the allocate function. The goal of, this function is to refill the address stack. To do this, the this function allocates another block of memory, divides the memory up, and adds it to the address stack:

```
void add_addrs()
{
    constexpr const auto block_size = 0x1000;
    auto block = std::make_unique<uint8_t[]>(block_size);

    auto v = gsl::span<uint8_t>(
        block.get(), block_size
    );

    auto total_size =
        v.size() % m_size == 0 ? v.size() : v.size() - m_size;

    for (auto i = 0; i < total_size; i += m_size)
    {
        m_addrs.push(&v.at(i));
    }

    m_blocks.push(std::move(block));
}
```

Finally, we have the private member variables, which includes the pool's size, the address stack, and the stack of blocks. Note that we use std::stack for this. std::stack uses std::deque to implement the stack, and although a more efficient stack can be written that doesn't leverage iterators, in testing, std::stack is nearly as performant:

```
size_type m_size;

std::stack<void *> m_addrs{};
std::stack<std::unique_ptr<uint8_t[]>> m_blocks{};
```

The allocator itself is nearly identical to the previous unequal allocator we already defined:

```
template<typename T>
class myallocator
{
public:

    using value_type = T;
    using pointer = T *;
    using size_type = std::size_t;
    using is_always_equal = std::false_type;
    using propagate_on_container_copy_assignment = std::false_type;
    using propagate_on_container_move_assignment = std::true_type;
    using propagate_on_container_swap = std::true_type;
```

One difference is that we define `propagate_on_container_copy_assignment` as `false`, specifically to prevent the allocator from being copied as much as possible. This choice is also backed by the fact that we already determined that glibc doesn't provide a huge benefit turning this property on when leveraging an unequal allocator.

The constructors are the same as previously defined:

```
    myallocator() :
        m_pool{std::make_shared<pool>(sizeof(T))}
    {
        std::cout << this << " constructor, sizeof(T): "
                  << sizeof(T) << '\n';
    }

    template <typename U>
    myallocator(const myallocator<U> &other) noexcept :
        m_pool{other.m_pool}
    {
        std::cout << this << " copy constructor (U), sizeof(T): "
                  << sizeof(T) << '\n';

        m_pool->rebind(sizeof(T));
    }

    myallocator(myallocator &&other) noexcept :
        m_pool{std::move(other.m_pool)}
    {
        std::cout << this << " move constructor, sizeof(T): "
                  << sizeof(T) << '\n';
    }

    myallocator &operator=(myallocator &&other) noexcept
    {
        std::cout << this << " move assignment, sizeof(T): "
                  << sizeof(T) << '\n';

        m_pool = std::move(other.m_pool);
        return *this;
    }

    myallocator(const myallocator &other) noexcept :
        m_pool{other.m_pool}
    {
        std::cout << this << " copy constructor, sizeof(T): "
                  << sizeof(T) << '\n';
    }

    myallocator &operator=(const myallocator &other) noexcept
    {
        std::cout << this << " copy assignment, sizeof(T): "
                  << sizeof(T) << '\n';

        m_pool = other.m_pool;
        return *this;
    }
```

The `allocate` and `deallocate` functions are the same as previously defined, calling the pool's allocation function. One difference is that our pool is only capable of allocating memory in single chunks (that is, the pool allocator is not capable of allocating more than one address while also preserving continuity). As a result, if `n` is something other than `1` (that is, the container is not `std::list` or `std::map`), we fall back to a `malloc()`/`free()` implementation, which is typically the default implementation:

```
        pointer allocate(size_type n)
        {
            if (n != 1) {
                return static_cast<pointer>(malloc(sizeof(T) * n));
            }

            return static_cast<pointer>(m_pool->allocate());
        }

        void deallocate(pointer ptr, size_type n)
        {
            if (n != 1) {
                free(ptr);
            }

            m_pool->deallocate(ptr);
        }
```

The rest of the allocator is the same:

```
private:

    std::shared_ptr<pool> m_pool;

    template <typename T1, typename T2>
    friend bool operator==(const myallocator<T1> &lhs, const
myallocator<T2> &rhs);

    template <typename T1, typename T2>
    friend bool operator!=(const myallocator<T1> &lhs, const
myallocator<T2> &rhs);

    template <typename U>
    friend class myallocator;
};

template <typename T1, typename T2>
bool operator==(const myallocator<T1> &lhs, const myallocator<T2> &rhs)
{ return lhs.m_pool.get() == rhs.m_pool.get(); }

template <typename T1, typename T2>
bool operator!=(const myallocator<T1> &lhs, const myallocator<T2> &rhs)
{ return lhs.m_pool.get() != rhs.m_pool.get(); }
```

Finally, before we can test our allocator, we will need to define a benchmarking function, capable of giving us an indication of how long a specific operation takes. This function will be defined in better detail in `Chapter 11`, *Time Interfaces in Unix*. For now, the most important thing to understand is that this function takes a callback function as an input (in our case, a Lambda), and returns a number. The higher the returned number, the longer the callback function took to execute:

```
template<typename FUNC>
auto benchmark(FUNC func) {
    auto stime = std::chrono::high_resolution_clock::now();
    func();
    auto etime = std::chrono::high_resolution_clock::now();

    return (etime - stime).count();
}
```

The first test we will perform is creating two lists and adding elements to each list, while timing how long it takes to add all of the elements to the list. Since each addition to the list requires an allocation, performing this test will give us a rough comparison on how much better our allocator is at allocating memory compared to the default allocator provided by glibc.

```
constexpr const auto num = 100000;

std::list<int> mylist1;
std::list<int, myallocator<int>> mylist2;

auto time1 = benchmark([&]{
    for (auto i = 0; i < num; i++) {
        mylist1.emplace_back(42);
    }
});

auto time2 = benchmark([&]{
    for (auto i = 0; i < num; i++) {
        mylist2.emplace_back(42);
    }
});

std::cout << "[TEST] add many:\n";
std::cout << " - time1: " << time1 << '\n';
std::cout << " - time2: " << time2 << '\n';
```

As stated, for each list, we add `100000` integers to the list and time how long it takes, giving us the ability to compare the allocators. The results are as follows:

```
0x7ffca71d7a00 constructor, sizeof(T): 24
[TEST] add many:
  - time1: 3921793
  - time2: 1787499
```

As shown, our allocator is 219% faster than the default allocator at allocating memory.

In our next test, we will compare our allocator with the default allocator with respect to deallocating memory. To perform this test, we will do the same thing as before, but instead of timing our allocations, we will time how long it takes to remove elements from each list:

```
constexpr const auto num = 100000;

std::list<int> mylist1;
std::list<int, myallocator<int>> mylist2;

for (auto i = 0; i < num; i++) {
    mylist1.emplace_back(42);
    mylist2.emplace_back(42);
}

auto time1 = benchmark([&]{
    for (auto i = 0; i < num; i++) {
        mylist1.pop_front();
    }
});

auto time2 = benchmark([&]{
    for (auto i = 0; i < num; i++) {
        mylist2.pop_front();
    }
});

std::cout << "[TEST] remove many:\n";
std::cout << " - time1: " << time1 << '\n';
std::cout << " - time2: " << time2 << '\n';
```

The results of the `this` function are as follows:

```
0x7fff14709720 constructor, sizeof(T): 24
[TEST] remove many:
  - time1: 1046463
  - time2: 1285248
```

As shown, our allocator is only 81% as fast as the default allocator. This is likely because the `free()` function is more efficient, which is not a surprise, as pushing to a stack could, in theory, be slower than some implementations of `free()`. Even though our `free()` function is slower, the difference is negligible compared to the improvement in both allocations and fragmentation. It is also important to note that the allocation and deallocation speeds are almost the same with this implementation, which is what we would expect.

To ensure we wrote our allocator correctly, the following will run our test again, but instead of timing how long it takes to add elements to the list, we will add up each value in the list. If our total is as we expect, we will know that allocations and deallocations were performed properly:

```
constexpr const auto num = 100000;

std::list<int, myallocator<int>> mylist;

for (auto i = 0; i < num; i++) {
    mylist.emplace_back(i);
}
```

```
    uint64_t total1{};
    uint64_t total2{};

    for (auto i = 0; i < num; i++) {
        total1 += i;
        total2 += mylist.back();
        mylist.pop_back();
    }

    std::cout << "[TEST] verify: ";
    if (total1 == total2) {
        std::cout << "success\n";
    }
    else {
        std::cout << "failure\n";
        std::cout << " - total1: " << total1 << '\n';
        std::cout << " - total2: " << total2 << '\n';
    }
```

As expected, the output of our test is `success`.

Compiling and testing

To compile this code, we leverage the same `CMakeLists.txt` file that we have been using for the other examples: `https://github.com/PacktPublishing/Hands-On-System-Programming-with-CPP/blob/master/Chapter09/CMakeLists.txt`.

With this code in place, we can compile this code using the following:

```
> git clone
https://github.com/PacktPublishing/Hands-On-System-Programming-with-CPP.git
> cd Hands-On-System-Programming-with-CPP/Chapter09/
> mkdir build
> cd build

> cmake -DCMAKE_BUILD_TYPE=Release ..
> make
```

To execute the example, run the following:

```
> ./example7
```

The output should resemble the following:

```
0x7ffca71d7a00 constructor, sizeof(T): 24
[TEST] add many:
  - time1: 3921793
  - time2: 1787499
0x7fff14709720 constructor, sizeof(T): 24
[TEST] remove many:
  - time1: 1046463
  - time2: 1285248
0x7fff5d8ad040 constructor, sizeof(T): 24
[TEST] verify: success
=============================================================================
====
test cases: 5 | 5 passed
assertions: - none -
```

As you can see, the output of our example matches the output we provided before. It should be noted that your results might very based on factors such as the hardware or what is already running on the box.

Summary

In this chapter, we looked at how to create our own allocators, and covered the intricate details of the C++ allocator concept. Topics included the difference between equal and unequal allocators, how container propagation is handled, rebinding, and potential issues with stateful allocators. Finally, we concluded with two different examples. The first example demonstrated how to create a simple, cache-aligned allocator that is stateless, while the second provided a functional example of a stateful object allocator that maintains a free pool for fast allocations.

In the next chapter, we will use several examples to demonstrate how to program POSIX sockets (that is, network programming) using C++.