7
Memory Management

After reading the previous chapters, it should no longer come as a surprise that the way we handle memory can have a huge impact on performance. The CPU spends a lot of time shuffling data between the CPU registers and the main memory (loading and storing data to and from the main memory). As shown in *Chapter 4*, *Data Structures*, the CPU uses memory caches to speed up access to memory, and programs need to be cache-friendly in order to run quickly.

This chapter will reveal more aspects of how computers work with memory so that you know which things must be considered when tuning memory usage. In addition, this chapter covers:

- Automatic memory allocation and dynamic memory management.
- The life cycle of a C++ object and how to manage object ownership.
- Efficient memory management. Sometimes, there are hard memory limits that force us to keep our data representation compact, and sometimes, we have plenty of memory available but need the program to go faster by making memory management more efficient.
- How to minimize dynamic memory allocations. Allocating and deallocating dynamic memory is relatively expensive and, at times, we need to avoid unnecessary allocations to make the program run faster.

We will start this chapter by explaining some concepts that you need to understand before we dig deeper into C++ memory management. This introduction will explain virtual memory and virtual address spaces, stack memory versus heap memory, paging, and swap space.

Computer memory

The physical memory of a computer is shared among all the processes running on a system. If one process uses a lot of memory, the other processes will most likely be affected. But from a programmer's perspective, we usually don't have to bother about the memory that is being used by other processes. This isolation of memory is due to the fact that most operating systems today are **virtual memory** operating systems, which provide the illusion that a process has all the memory for itself. Each process has its own **virtual address space**.

The virtual address space

Addresses in the virtual address space that programmers see are mapped to physical addresses by the operating system and the **memory management unit** (**MMU**), which is a part of the processor. This mapping or translation happens each time we access a memory address.

This extra layer of indirection makes it possible for the operating system to use physical memory for the parts of a process that are currently being used, and back up the rest of the virtual memory on disk. In this sense, we can look at the physical main memory as a cache for the virtual memory space, which resides on secondary storage. The areas of the secondary storage that are used for backing up memory pages are usually called **swap space**, **swap file**, or simply **pagefile**, depending on the operating system.

Virtual memory makes it possible for processes to have a virtual address space bigger than the physical address space, since virtual memory that is not in use does not have to occupy physical memory.

Memory pages

The most common way to implement virtual memory today is to divide the address space into fixed-size blocks called **memory pages**. When a process accesses memory at a virtual address, the operating system

checks whether the memory page is backed by physical memory (a page frame). If the memory page is not mapped in the main memory, a hardware exception occurs, and the page is loaded from disk into memory. This type of hardware exception is called a **page fault**. This is not an error but a necessary interrupt in order to load data from disk to memory. As you may have guessed, though, this is very slow compared to reading data that is already resident in memory.

When there are no more available page frames in the main memory, a page frame has to be evicted. If the page to be evicted is dirty, that is, it has been modified since it was last loaded from disk, it needs to be written to disk before it can be replaced. This mechanism is called **paging**. If the memory page has not been modified, the memory page is simply evicted.

Not all operating systems that support virtual memory support paging. iOS, for example, does have virtual memory but dirty pages are never stored on disk; only clean pages can be evicted from memory. If the main memory is full, iOS will start terminating processes until there is enough free memory again. Android uses a similar strategy. One reason for not writing memory pages back to the flash storage of the mobile devices is that it drains the battery, and it also shortens the lifespan of the flash storage itself.

The following diagram shows two running processes. They both have their own virtual memory space. Some of the pages are mapped to the physical memory, while some are not. If process 1 needs to use memory in the memory page that starts at address 0x1000, a page fault will occur. The memory page will then be mapped to a vacant memory frame. Also, note that the virtual memory addresses are not the same as the physical addresses. The first memory page of process 1, which starts at the virtual address 0x0000, is mapped to a memory frame that starts at the physical address 0x4000:

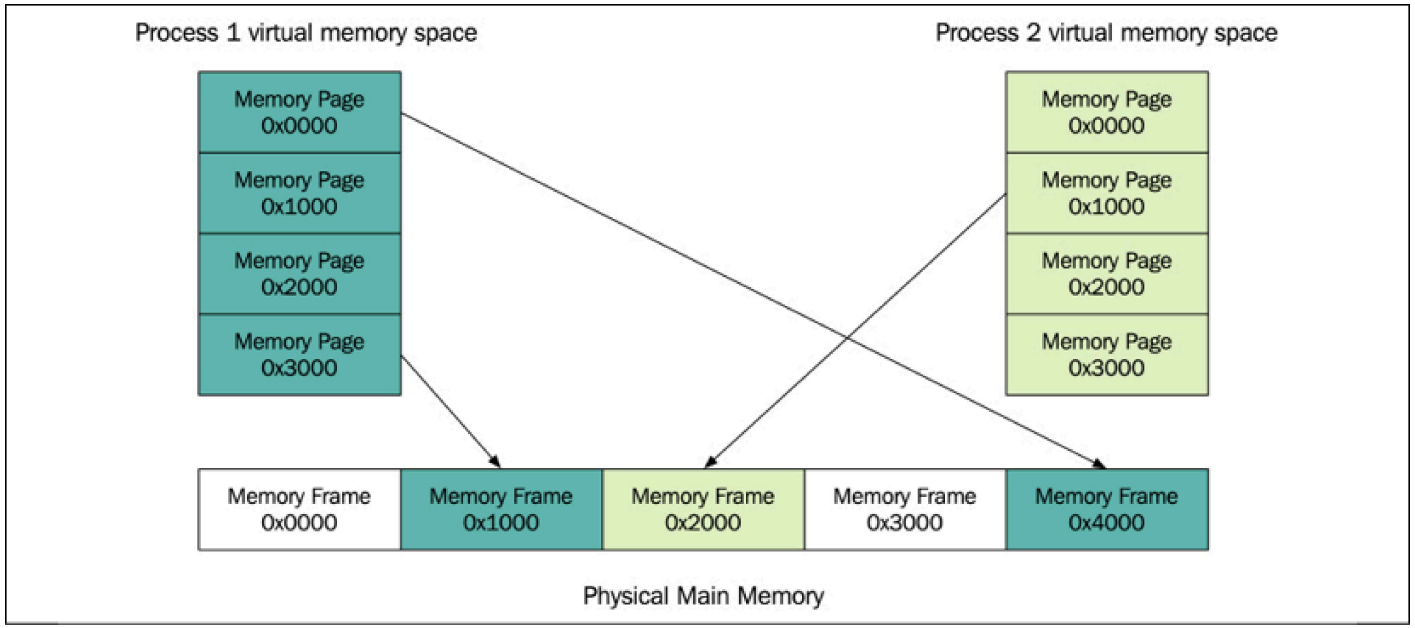

Figure 7.1: Virtual memory pages, mapped to memory frames in physical memory. Virtual memory pages that are not in use do not have to occupy physical memory.

Thrashing

Thrashing can happen when a system runs low on physical memory and is, therefore, constantly paging. Whenever a process gets time scheduled on the CPU, it tries to access memory that has been paged out. Loading new memory pages means that the other pages first have to be stored on disk. Moving data back and forth between disk and memory is usually very slow; in some cases, this more or less stalls the computer since the system spends all its time paging. Looking at the system's page fault frequency is a good way to determine whether the program has started thrashing.

Knowing the basics of how memory is being handled by the hardware and the OS is important when optimizing performance. Next, we will see how memory is handled during the execution of a C++ program.

Process memory

The stack and the heap are the two most important memory segments in a C++ program. There is also static storage and thread local storage, but we will talk more about that later. Actually, to be formally correct, C++ doesn't talk about stack and heap; instead, it talks about the free store, storage classes, and the storage duration of objects. However, since the concepts of stack and heap are widely used in the C++ community, and all the implementations of C++ that we are aware of use a stack to implement function calls and manage the automatic storage of local variables, it is important to understand what stack and heap are.

In this book, I will also use the terms *stack* and *heap* rather than the storage duration of objects. I will use the terms *heap* and *free store* interchangeably and will not make any distinction between them.

Both the stack and the heap reside in the process' virtual memory space. The stack is a place where all the local variables reside; this also includes arguments to functions. The stack grows each time a function is called and contracts when a function returns. Each thread has its own stack and, hence, stack memory can be considered thread-safe. The heap, on the other hand, is a global memory area that is shared among all the threads in a running process. The heap grows when we allocate memory with `new` (or the C library functions `malloc()` and `calloc()`) and contracts when we free the memory with `delete` (or `free()`). Usually, the heap starts at a low address and grows in an upward direction, whereas the stack starts at a high address and grows in a downward direction. *Figure 7.2* shows how the stack and heap grow in opposite directions in a virtual address space:

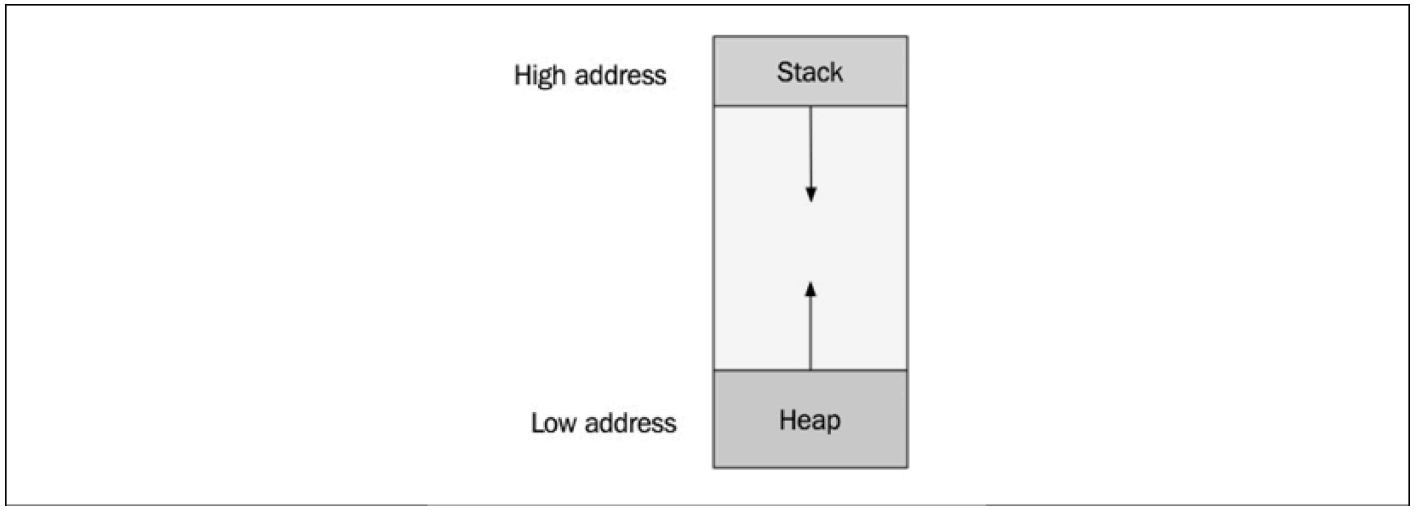

Figure 7.2: An address space of a process. The stack and the heap grow in opposite directions.

The next sections will provide more details about the stack and the heap, and also explain when we are using each of these memory areas in the C++ programs we write.

Stack memory

The stack differs in many ways compared to the heap. Here are some of the unique properties of the stack:

- The stack is a contiguous memory block.
- It has a fixed maximum size. If a program exceeds the maximum stack size, the program will crash. This condition is called stack overflow.
- The stack memory never becomes fragmented.
- Allocating memory from the stack is (almost) always fast. Page faults are possible but rare.
- Each thread in a program has its own stack.

The code examples that follow in this section will examine some of these properties. Let's start with allocations and deallocations to get a feel for how the stack is used in a program.

We can easily find out in which direction the stack grows by inspecting the address of the stack-allocated data. The following example code demonstrates how the stack grows and contracts when entering and leaving functions:

```
void func1() {
  auto i = 0;
  std::cout << "func1(): " << std::addressof(i) << '\n';
}
void func2() {
  auto i = 0;
  std::cout << "func2(): " << std::addressof(i) << '\n';
  func1();
}

int main() {
  auto i = 0;
  std::cout << "main():  " << std::addressof(i) << '\n';
  func2();
  func1();
}
```

A possible output when running the program could look like this:

```
main():  0x7ea075ac
func2(): 0x7ea07594
func1(): 0x7ea0757c
func1(): 0x7ea07594
```

By printing the address of the stack allocated integer, we can determine how much and in which direction the stack grows on my platform. The stack grows by 24 bytes each time we enter either `func1()` or `func2()`. The integer `i`, which will be allocated on the stack, is 4 bytes long. The remaining 20 bytes contain data needed when the function ends, such as the return address, and perhaps some padding for alignment.

The following diagram illustrates how the stack grows and contracts during the execution of the program. The first box illustrates how the memory looks when the program has just entered the `main()` function. The second box shows how the stack has increased when we execute `func1()`, and so on:

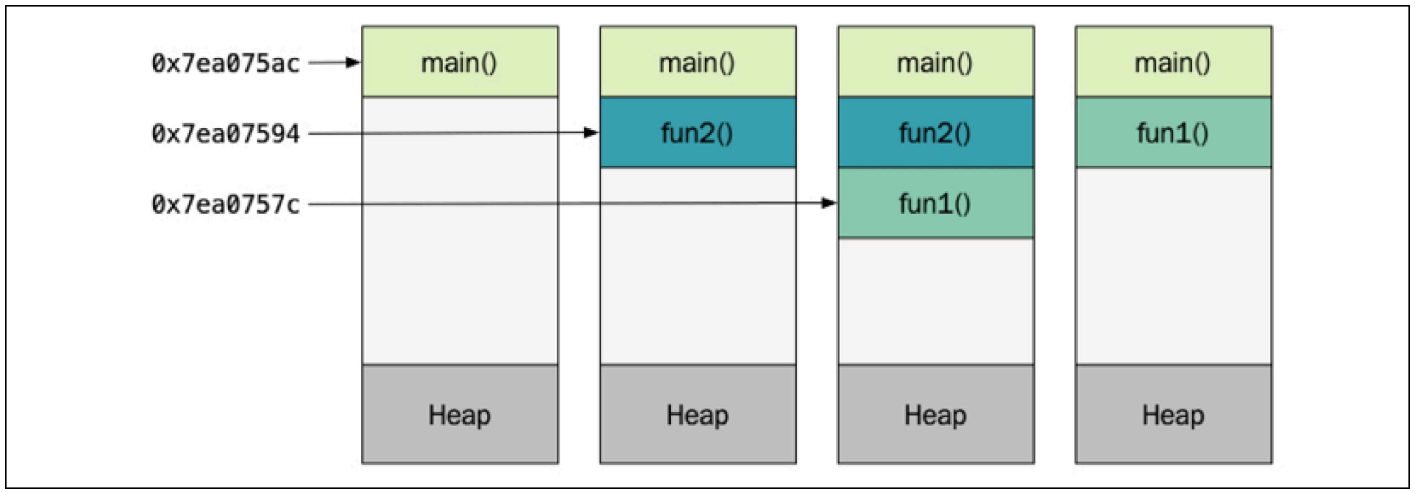

Figure 7.3: The stack grows and contracts when functions are entered

The total memory allocated for the stack is a fixed-size contiguous memory block created at thread startup. So, how big is the stack and what happens when we reach the limit of the stack?

As mentioned earlier, the stack grows each time the program enters a function and contracts when the function returns. The stack also grows whenever we create a new stack variable within the same function and contracts when such a variable goes out of scope. The most common reason for the stack to overflow

is by deep recursive calls and/or by using large, automatic variables on the stack. The maximum size of the stack differs among platforms and can also be configured for individual processes and threads.

Let's see if we can write a program to see how big the stack is by default on my system. We will begin by writing a function, `func()`, which will recurse infinitely. At the beginning of each function, we'll allocate a 1-kilobyte variable, which will be placed onto the stack every time we enter `func()`. Every time `func()` is executed, we print the current size of the stack:

```
void func(std::byte* stack_bottom_addr) {
  std::byte data[1024];
  std::cout << stack_bottom_addr - data << '\n';
  func(stack_bottom_addr);
}

int main() {
  std::byte b;
  func(&b);
}
```

The size of the stack is only an estimate. We compute it by subtracting the address of the first local variable in `main()` from the first local variable defined in `func()`.

When I compiled the code with Clang, I got a warning that `func()` will never return. Normally, this is a warning that we should not ignore, but this time, this is exactly the result we want, so we ignore the warning and run the program anyway. The program crashes after a short while when the stack has reached its limit. Before the program crashes, it manages to print out thousands of lines with the current size of the stack. The last lines of the output look like this:

```
...
8378667
8379755
8380843
```

Since we are subtracting `std::byte` pointers, the size is in bytes, so it looks like the maximum size of the stack is around 8 MB on my system. On Unix-like systems, it is possible to set and get the stack size for processes by using the `ulimit` command with the option `-s`:

```
$ ulimit -s
$ 8192
```

`ulimit` (short for user limit) returns the current setting for the maximum stack size in kilobytes. The output of `ulimit` confirms the results from our experiment: the stack is about 8 MB on my Mac if I don't configure it explicitly.

On Windows, the default stack size is usually set to 1 MB. A program running fine on macOS might crash due to a stack overflow on Windows if the stack size is not correctly configured.

With this example, we can also conclude that we don't want to run out of stack memory since the program will crash when that happens. Later in this chapter, we will see how to implement a rudimentary memory allocator to handle fixed-size allocations. We will then understand that the stack is just another type of memory allocator that can be implemented very efficiently because the usage pattern is always sequential. We always request and release memory at the top of the stack (the end of the contiguous memory). This ensures that the stack memory will never become fragmented and that we can allocate and deallocate memory by only moving a stack pointer.

Heap memory

The heap (or the free store, which is a more correct term in C++) is where data with dynamic storage lives. As mentioned earlier, the heap is shared among multiple threads, which means that memory management for the heap needs to take concurrency into account. This makes memory allocations in the heap more complicated than stack allocations, which are local per thread.

The allocation and deallocation pattern for stack memory is sequential, in the sense that memory is always deallocated in the reverse order to that in which it was allocated. On the other hand, for dynamic memory, the allocations and deallocations can happen arbitrarily. The dynamic lifetime of objects and the variable sizes of memory allocations increase the risk of **fragmented memory**.

An easy way to understand the issue with memory fragmentation is to go through an example of how fragmented memory can occur. Suppose that we have a small contiguous memory block of 16 KB that we are allocating memory from. We are allocating objects of two types: type **A**, which is 1 KB, and type **B**, which is 2 KB. We first allocate an object of type **A**, followed by an object of type **B**. This repeats until the memory looks like the following image:

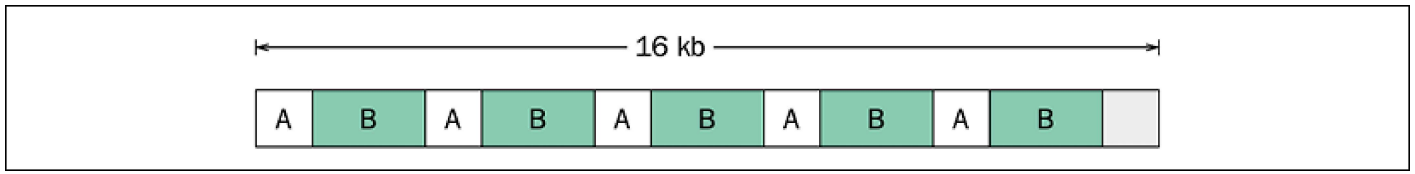

Figure 7.4: The memory after allocating objects of type A and B

Next, all objects of type **A** are no longer needed, so they can be deallocated. The memory now looks like this:

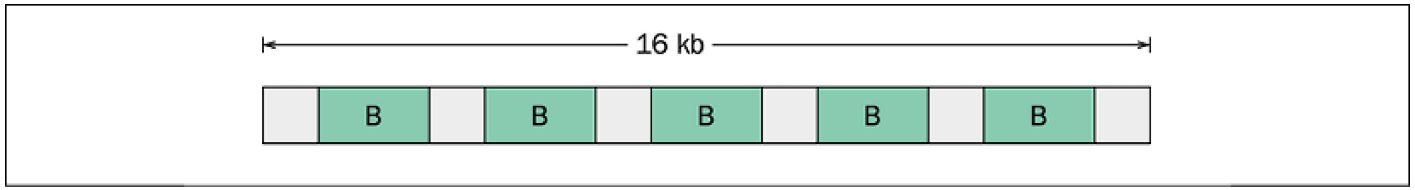

Figure 7.5: The memory after objects of type A are deallocated

There is now 10 KB of memory in use and 6 KB is available. Now, suppose we want to allocate a new object of type **B**, which is 2 KB. Although there is 6 KB of free memory, there is nowhere we can find a 2 KB memory block because the memory has become fragmented.

Now that you have a good understanding of how computer memory is structured and used in a running process, it's time to explore how C++ objects live in memory.

Objects in memory

All the objects we use in a C++ program reside in memory. Here, we will explore how objects are created and deleted from memory, and also describe how objects are laid out in memory.

Creating and deleting objects

In this section, we will dig into the details of using `new` and `delete`. Consider the following way of using `new` to create an object on the free store and then deleting it using `delete`:

```
auto* user = new User{"John"};  // allocate and construct
user->print_name();             // use object
delete user;                    // destruct and deallocate
```

I don't recommend that you call `new` and `delete` explicitly in this manner, but let's ignore that for now. Let's get to the point; as the comments suggest, `new` actually does two things, namely:

- Allocates memory to hold a new object of the `User` type
- Constructs a new `User` object in the allocated memory space by calling the constructor of the `User` class

The same thing goes with `delete`, it:

- Destructs the `User` object by calling its destructor
- Deallocates/frees the memory that the `User` object was placed in

It is actually possible to separate these two actions (memory allocation and object construction) in C++. This is rarely used but has some important and legitimate use cases when writing library components.

Placement new

C++ allows us to separate memory allocation from object construction. We could, for example, allocate a byte array with `malloc()` and construct a new `User` object in that region of memory. Have a look at the following code snippet:

```
auto* memory = std::malloc(sizeof(User));
auto* user = ::new (memory) User("john");
```

The perhaps unfamiliar syntax that's using `::new (memory)` is called **placement new**. It is a non-allocating form of `new`, which only constructs an object. The double colon (`::`) in front of `new` ensures that the resolution occurs from the global namespace to avoid picking up an overloaded version of `operator new`.

In the preceding example, placement new constructs the `User` object and places it at the specified memory location. Since we are allocating the memory with `std::malloc()` for a single object, it is guaranteed to be correctly aligned (unless the class `User` has been declared to be overaligned). Later on, we will explore cases where we have to take alignment into account when using placement new.

There is no placement delete, so in order to destruct the object and free the memory, we need to call the destructor explicitly and then free the memory:

```
user->~User();
std::free(memory);
```

This is the only time you should call a destructor explicitly. Never call a destructor like this unless you have created an object with placement new.

C++17 introduces a set of utility functions in `<memory>` for constructing and destroying objects without allocating or deallocating memory. So, instead of calling placement new, it is now possible to use some of the functions from `<memory>` whose names begin with `std::uninitialized_` for constructing, copying, and moving objects to an uninitialized memory area. And instead of calling the destructor explicitly, we can now use `std::destroy_at()` to destruct an object at a specific memory address without deallocating the memory.

The previous example could be rewritten using these new functions. Here is how it would look:

```
auto* memory = std::malloc(sizeof(User));
auto* user_ptr = reinterpret_cast<User*>(memory);
std::uninitialized_fill_n(user_ptr, 1, User{"john"});
std::destroy_at(user_ptr);
std::free(memory);
```

C++20 also introduces `std::construct_at()`, which makes it possible to replace the `std::uninitialized_fill_n()` call with:

```
std::construct_at(user_ptr, User{"john"});      // C++20
```

Please keep in mind that we are showing these naked low-level memory facilities to get a better understanding of memory management in C++. Using `reinterpret_cast` and the memory utilities demonstrated here should be kept to an absolute minimum in a C++ code base.

Next, you will see what operators are called when we use the `new` and `delete` expressions.

The new and delete operators

The function `operator new` is responsible for allocating memory when a new expression is invoked. The `new` operator can be either a globally defined function or a static member function of a class. It is possible to overload the global operators `new` and `delete`. Later in this chapter, we will see that this can be useful when analyzing memory usage.

Here is how to do it:

```
auto operator new(size_t size) -> void* {
  void* p = std::malloc(size);
  std::cout << "allocated " << size << " byte(s)\n";
  return p;
}

auto operator delete(void* p) noexcept -> void {
  std::cout << "deleted memory\n";
```

```
  return std::free(p);
}
```

We can verify that our overloaded operators are actually being used when creating and deleting a `char` object:

```
auto* p = new char{'a'}; // Outputs "allocated 1 byte(s)"
delete p;                // Outputs "deleted memory"
```

When creating and deleting an array of objects using the `new[]` and `delete[]` expressions, there is another pair of operators that are being used, namely `operator new[]` and `operator delete[]`. We can overload these operators in the same way:

```
auto operator new[](size_t size) -> void* {
  void* p = std::malloc(size);
  std::cout << "allocated " << size << " byte(s) with new[]\n";
  return p;
}

auto operator delete[](void* p) noexcept -> void {
  std::cout << "deleted memory with delete[]\n";
  return std::free(p);
}
```

Keep in mind that if you overload `operator new`, you should also overload `operator delete`. Functions for allocating and deallocating memory come in pairs. Memory should be deallocated by the allocator that the memory was allocated by. For example, memory allocated with `std::malloc()` should always be

freed using `std::free()`, while memory allocated with `operator new[]` should be deallocated using `operator delete[]`.

It is also possible to override a class-specific `operator new` or `operator delete`. This is probably more useful than overloading the global operators, since it is more likely that we need a custom dynamic memory allocator for a specific class.

Here, we are overloading `operator new` and `operator delete` for the `Document` class:

```
class Document {
// ...
public:
  auto operator new(size_t size) -> void* {
    return ::operator new(size);
  }
  auto operator delete(void* p) -> void {
    ::operator delete(p);
  }
};
```

The class-specific version of `new` will be used when we create new dynamically allocated `Document` objects:

```
auto* p = new Document{}; // Uses class-specific operator new
delete p;
```

If we instead want to use global `new` and `delete`, it is still possible by using the global scope (`::`):

```
auto* p = ::new Document{}; // Uses global operator new
::delete p;
```

We will discuss memory allocators later in this chapter and we will then see the overloaded `new` and `delete` operators in use.

To summarize what we have seen so far, a `new` expression involves two things: allocation and construction. `operator new` allocates memory and you can overload it globally or per class to customize dynamic memory management. Placement new can be used to construct an object in an already allocated memory area.

Another important, but rather low-level, topic that we need to understand in order to use memory efficiently is the **alignment** of memory.

Memory alignment

The CPU reads memory into its registers one word at a time. The word size is 64 bits on a 64-bit architecture, 32 bits on a 32-bit architecture, and so forth. For the CPU to work efficiently when working with different data types, it has restrictions on the addresses where objects of different types are located. Every type in C++ has an alignment requirement that defines the addresses at which an object of a certain type should be located in memory.

If the alignment of a type is 1, it means that the objects of that type can be located at any byte address. If the alignment of a type is 2, it means that the number of bytes between successive allowed addresses is 2. Or to quote the C++ standard:

> *"An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated."*

We can use `alignof` to find out the alignment of a type:

```
// Possible output is 4
std::cout << alignof(int) << '\n';
```

When I run this code, it outputs `4`, which means that the alignment requirement of the type `int` is 4 bytes on my platform.

The following figure shows two examples of memory from a system with 64-bit words. The upper row contains three 4-byte integers, which are located on addresses that are 4 bytes aligned. The CPU can load these integers into registers in an efficient way and never need to read multiple words when accessing one of the `int` members. Compare this with the second row, which contains two `int` members, which are located at unaligned addresses. The second `int` even spans over two-word boundaries. In the best case, this is just inefficient, but on some platforms, the program will crash:

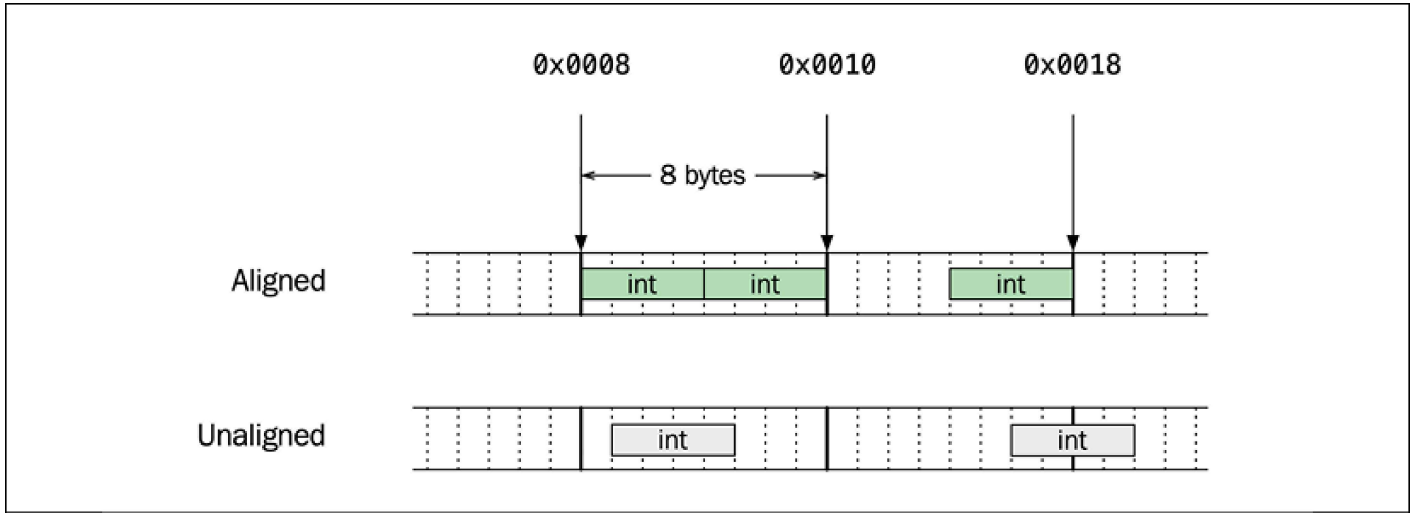

Figure 7.6: Two examples of memory that contain ints at aligned and unaligned memory addresses

Let's say that we have a type with an alignment requirement of 2. The C++ standard doesn't say whether the valid addresses are 1, 3, 5, 7... or 0, 2, 4, 6.... All platforms that we are aware of start counting addresses at 0, so, in practice we could check if an object is correctly aligned by checking if its address is a multiple of the alignment using the modulo operator (`%`).

However, if we want to write fully portable C++ code, we need to use `std::align()` and not modulo to check the alignment of an object. `std::align()` is a function from `<memory>` that will adjust a pointer according to an alignment that we pass as an argument. If the memory address we pass to it is already aligned, the pointer will not be adjusted. Therefore, we can use `std::align()` to implement a small utility function called `is_aligned()`, as follows:

```
bool is_aligned(void* ptr, std::size_t alignment) {
  assert(ptr != nullptr);
  assert(std::has_single_bit(alignment)); // Power of 2
  auto s = std::numeric_limits<std::size_t>::max();
  auto aligned_ptr = ptr;
  std::align(alignment, 1, aligned_ptr, s);
  return ptr == aligned_ptr;
}
```

At first, we make sure that the `ptr` argument isn't null and that `alignment` is a power of 2, which is stated as a requirement in the C++ standard. We are using C++20 `std::has_single_bit()` from the `<bit>` header to check this. Next, we are calling `std::align()`. The typical use case for `std::align()` is when we have a memory buffer of some size in which we want to store an object with some alignment requirement. In this case, we don't have a buffer, and we don't care about the size of the objects, so we say that the object is of size 1 and the buffer is the maximum value of a `std::size_t`. Then, we can compare the original `ptr` and the adjusted `aligned_ptr` to see if the original pointer was already aligned. We will have use for this utility in the examples to come.

When allocating memory with `new` or `std::malloc()`, the memory we get back should be correctly aligned for the type we specify. The following code shows that the memory allocated for `int` is at least 4 bytes aligned on my platform:

```
auto* p = new int{};
assert(is_aligned(p, 4ul)); // True
```

In fact, `new` and `malloc()` are guaranteed to always return memory suitably aligned for any scalar type (if it manages to return memory at all). The `<cstddef>` header provides us with a type called `std::max_align_t`, whose alignment requirement is at least as strict as all the scalar types. Later on, we will see that this type is useful when writing custom memory allocators. So, even if we only request memory for `char` on the free store, it will be aligned suitably for `std::max_align_t`.

The following code shows that the memory returned from `new` is correctly aligned for `std::max_align_t` and also for any scalar type:

```
auto* p = new char{};
auto max_alignment = alignof(std::max_align_t);
assert(is_aligned(p, max_alignment)); // True
```

Let's allocate `char` two times in a row with `new`:

```
auto* p1 = new char{'a'};
auto* p2 = new char{'b'};
```

Then, the memory may look something like this:

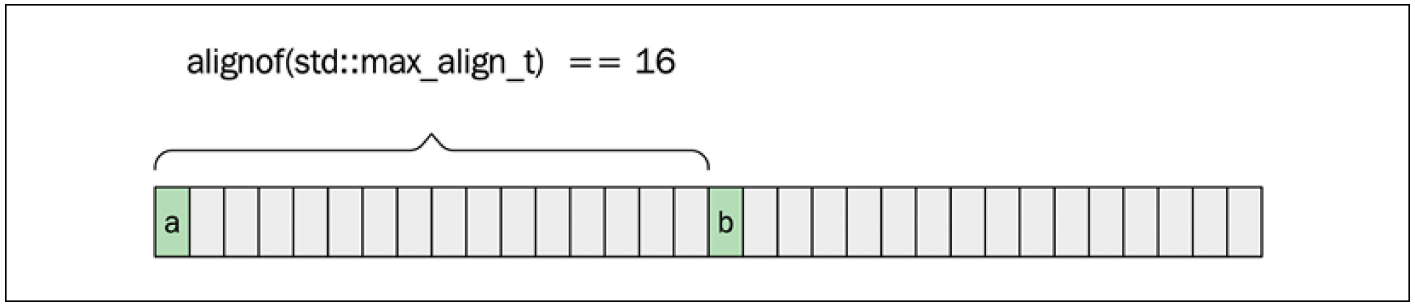

Figure 7.7: Memory layout after two separate allocations of one char each

The space between `p1` and `p2` depends on the alignment requirements of `std::max_align_t`. On my system, it was `16` bytes and, therefore, there are 15 bytes between each `char` instance, even though the alignment of a `char` is only 1.

It is possible to specify custom alignment requirements that are stricter than the default alignment when declaring a variable using the `alignas` specifier. Let's say we have a cache line size of 64 bytes and that we, for some reason, want to ensure that two variables are placed on separate cache lines. We could do the following:

```
alignas(64) int x{};
alignas(64) int y{};
// x and y will be placed on different cache lines
```

It's also possible to specify a custom alignment when defining a type. The following is a struct that will occupy exactly one cache line when being used:

```
struct alignas(64) CacheLine {
   std::byte data[64];
};
```

Now, if we were to create a stack variable of the type `CacheLine`, it would be aligned according to the custom alignment of 64 bytes:

```
int main() {
  auto x = CacheLine{};
  auto y = CacheLine{};
  assert(is_aligned(&x, 64));
  assert(is_aligned(&y, 64));
  // ...
}
```

The stricter alignment requirements are also satisfied when allocating objects on the heap. In order to support dynamic allocation of types with non-default alignment requirements, C++17 introduced new overloads of `operator new()` and `operator delete()` which accept an alignment argument of type `std::align_val_t`. There is also an `aligned_alloc()` function defined in `<cstdlib>` which can be used to manually allocate aligned heap memory.

As follows is an example in which we allocate a block of heap memory that should occupy exactly one memory page. In this case, the alignment-aware versions of `operator new()` and `operator delete()` will be invoked when using `new` and `delete`:

```
constexpr auto ps = std::size_t{4096};     // Page size
struct alignas(ps) Page {
   std::byte data_[ps];
};
auto* page = new Page{};                   // Memory page
assert(is_aligned(page, ps));              // True
// Use page ...
delete page;
```

Memory pages are not part of the C++ abstract machine, so there is no portable way to programmatically get hold of the page size of the currently running system. However, you could use `boost::mapped_region::get_page_size()` or a platform-specific system call, such as `getpagesize()`, on Unix systems.

A final caveat to be aware of is that the supported set of alignments are defined by the implementation of the standard library you are using, and not the C++ standard.

Padding

The compiler sometimes needs to add extra bytes, **padding**, to our user-defined types. When we define data members in a class or struct, the compiler is forced to place the members in the same order as we define them.

However, the compiler also has to ensure that the data members inside the class have the correct alignment; hence, it needs to add padding between data members if necessary. For example, let's assume we have a class defined as follows:

```
class Document {
  bool is_cached_{};
  double rank_{};
  int id_{};
};
std::cout << sizeof(Document) << '\n'; // Possible output is 24
```

The reason for the possible output being 24 is that the compiler inserts padding after `bool` and `int`, to fulfill the alignment requirements of the individual data members and the entire class. The compiler converts the `Document` class into something like this:

```
class Document {
  bool is_cached_{};
  std::byte padding1[7]; // Invisible padding inserted by compiler
  double rank_{};
  int id_{};
  std::byte padding2[4]; // Invisible padding inserted by compiler
};
```

The first padding between `bool` and `double` is 7 bytes, since the `rank_` data member of the `double` type has an alignment of 8 bytes. The second padding that is added after `int` is 4 bytes. This is needed in order to fulfill the alignment requirements of the `Document` class itself. The member with the largest alignment requirement also determines the alignment requirement for the entire data structure. In our example, this means that the total size of the `Document` class must be a multiple of 8, since it contains a `double` value that is 8-byte aligned.

We now realize that we can rearrange the order of the data members in the `Document` class in a way that minimizes the padding inserted by the compiler, by starting with types with the biggest alignment requirements. Let's create a new version of the `Document` class:

```
// Version 2 of Document class
class Document {
  double rank_{}; // Rearranged data members
  int id_{};
  bool is_cached_{};
};
```

With the rearrangement of the members, the compiler now only needs to pad after the `is_cached_` data member to adjust for the alignment of `Document`. This is how the class will look after padding:

```
// Version 2 of Document class after padding
class Document {
  double rank_{};
  int id_{};
  bool is_cached_{};
  std::byte padding[3]; // Invisible padding inserted by compiler
};
```

The size of the new `Document` class is now only 16 bytes, compared to the first version, which was 24 bytes. The insight here should be that the size of an object can change just by changing the order in which its members are declared. We can also verify this by using the `sizeof` operator again on our updated version of `Document`:

```
std::cout << sizeof(Document) << '\n'; // Possible output is 16
```

The following image shows the memory layout of version 1 and version 2 of the `Document` class:

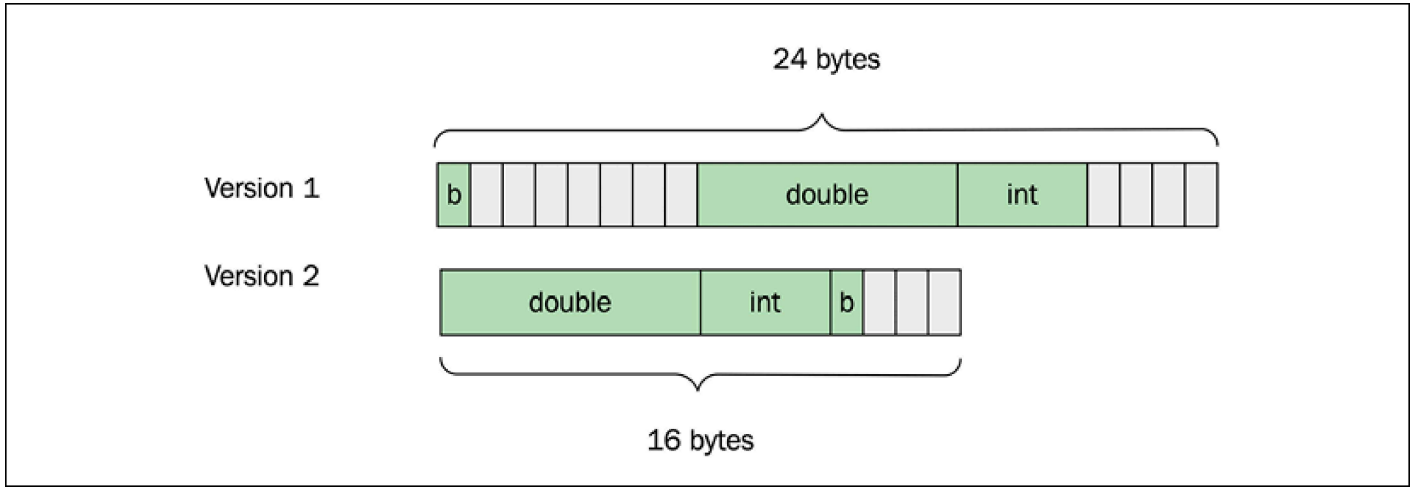

Figure 7.8: Memory layouts of the two versions of the Document class. The size of an object can change just by changing the order in which its members are declared.

As a general rule, you can place the biggest data members at the beginning and the smallest members at the end. In this way, you can minimize the memory overhead caused by padding. Later on, we will see that we need to think about alignment when placing objects in memory regions that we have allocated, before we know the alignment of the objects that we are creating.

From a performance perspective, there can also be cases where you want to align objects to cache lines to minimize the number of cache lines an object spans over. While we are on the subject of cache friendliness, it should also be mentioned that it can be beneficial to place multiple data members that are frequently used together next to each other.

Keeping your data structures compact is important for performance. Many applications are bound by memory access time. Another important aspect of memory management is to never leak or waste memory for objects that are no longer needed. We can effectively avoid all sorts of resource leaks by being clear and explicit about the ownership of resources. This is the topic of the following section.

Memory ownership

Ownership of resources is a fundamental aspect to consider when programming. An owner of a resource is responsible for freeing the resource when it is no longer needed. A resource is typically a block of memory but could also be a database connection, a file handle, and so on. Ownership is important, regardless of which programming language you are using. However, it is more apparent in languages such as C and C++, since dynamic memory is not garbage-collected by default. Whenever we allocate dynamic memory in C++, we have to think about the ownership of that memory. Fortunately, there is now very good support in the language for expressing various types of ownership by using smart pointers, which we will cover later in this section.

The smart pointers from the standard library help us specify the ownership of dynamic variables. Other types of variables already have a defined ownership. For example, local variables are owned by the current scope. When the scope ends, the objects that have been created inside the scope will be automatically destroyed:

```
{
  auto user = User{};
} // user automatically destroys when it goes out of scope
```

Static and global variables are owned by the program and will be destroyed when the program terminates:

```
static auto user = User{};
```

Data members are owned by the instances of the class that they belong to:

```
class Game {
  User user; // A Game object owns the User object
  // ...
};
```

It is only dynamic variables that do not have a default owner, and it is up to the programmer to make sure that all the dynamically allocated variables have an owner to control the lifetime of the variables:

```
auto* user = new User{}; // Who owns user now?
```

With modern C++, we can write most of our code without explicit calls to `new` and `delete`, which is a great thing. Manually keeping track of calls to `new` and `delete` can very easily become an issue with memory leaks, double deletes, and other nasty bugs as a result. Raw pointers do not express any ownership, which makes ownership hard to track if we are only using raw pointers to refer to dynamic memory.

I recommend that you make ownership clear and explicit, but do strive to minimize manual memory management. By following a few fairly simple rules for dealing with the ownership of memory, you will increase the likelihood of getting your code clean and correct without leaking resources. The coming sections will guide you through some best practices for that purpose.

Handling resources implicitly

First, make your objects implicitly handle the allocation/deallocation of dynamic memory:

```
auto func() {
  auto v = std::vector<int>{1, 2, 3, 4, 5};
}
```

In the preceding example, we are using both stack and dynamic memory, but we don't have to explicitly call `new` and `delete`. The `std::vector` object we create is an automatic object that will live on the stack. Since it is owned by the scope, it will be automatically destroyed when the function returns. The `std::vector` object itself uses dynamic memory to store the integer elements. When `v` goes out of scope, its destructor can safely free the dynamic memory. This pattern of letting destructors free dynamic memory makes it fairly easy to avoid memory leaks.

While we are on the subject of freeing resources, I think it makes sense to mention RAII. **RAII** is a well-known C++ technique, short for **Resource Acquisition Is Initialization**, where the lifetime of a resource is controlled by the lifetime of an object. The pattern is simple but extremely useful for handling resources

(memory included). But let's say, for a change, that the resource we need is some sort of connection for sending requests. Whenever we are done using the connection, we (the owners) must remember to close it. Here is an example of how it looks when we open and close the connection manually to send a request:

```
auto send_request(const std::string& request) {
  auto connection = open_connection("http://www.example.com/");
  send_request(connection, request);
  close(connection);
}
```

As you can see, we have to remember to close the connection after we have used it, or the connection will stay open (leak). In this example, it seems hard to forget, but once the code gets more complicated after inserting proper error handling and multiple exit paths, it will be hard to guarantee that the connection will always be closed. RAII solves this by relying on the fact that the lifetime of automatic variables is handled for us in a predictable way. What we need is an object that will have the same lifetime as the connection we get from the `open_connection()` call. We can create a class for this, called `RAIIConnection`:

```
class RAIIConnection {
public:
  explicit RAIIConnection(const std::string& url)
      : connection_{open_connection(url)} {}
  ~RAIIConnection() {
    try {
      close(connection_);
    }
    catch (const std::exception&) {
      // Handle error, but never throw from a destructor
    }
```

```
  }
  auto& get() { return connection_; }

private:
  Connection connection_;
};
```

The `Connection` object is now wrapped in a class that controls the lifetime of the connection (the resource). Instead of manually closing the connection, we can now let `RAIIConnection` handle this for us:

```
auto send_request(const std::string& request) {
  auto connection = RAIIConnection("http://www.example.com/");
  send_request(connection.get(), request);
  // No need to close the connection, it is automatically handled
  // by the RAIIConnection destructor
}
```

RAII makes our code safer. Even if `send_request()` would throw an exception here, the connection object would still be destructed and close the connection. We can use RAII for many types of resources, not just memory, file handles, and connections. Another example is `std::scoped_lock` from the C++ standard library. It tries to acquire a lock (mutex) on creation and then releases the lock on destruction. You can read more about `std::scoped_lock` in *Chapter 11*, *Concurrency*.

Now, we will explore more ways to make memory ownership explicit in C++.

Containers

You can use standard containers to handle a collection of objects. The container you use will own the dynamic memory it needs to store the objects you add to it. This is a very effective way of minimizing man-

ual `new` and `delete` expressions in your code.

It's also possible to use `std::optional` to handle the lifetime of an object that might or might not exist. `std::optional` can be seen as a container with a maximum size of 1.

We won't talk more about the containers here, since they have already been covered in *Chapter 4*, *Data Structures*.

Smart pointers

The smart pointers from the standard library wrap a raw pointer and make the ownership of the object it points to explicit. When used correctly, there is no doubt about who is responsible for deleting a dynamic object. The three smart pointer types are: `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`. As their names suggest, they represent three types of ownership of an object:

- Unique ownership expresses that I, and only I, own the object. When I'm done using it, I will delete it.
- Shared ownership expresses that I own the object along with others. When no one needs the object anymore, it will be deleted.
- Weak ownership expresses that I'll use the object if it exists, but don't keep it alive just for me.

We'll deal with each of these types, respectively, in the following sections.

Unique pointer

The safest and least complicated ownership is unique ownership and should be the first thing that pops into your mind when thinking about smart pointers. Unique pointers represent unique ownership; that is, a resource is owned by exactly one entity. Unique ownership can be transferred to someone else, but it cannot be copied, since that would break its uniqueness. Here is how to use a `std::unique_ptr`:

```
auto owner = std::make_unique<User>("John");
auto new_owner = std::move(owner); // Transfer ownership
```

Unique pointers are also very efficient since they add very little performance overhead compared to ordinary raw pointers. The slight overhead is incurred by the fact that `std::unique_ptr` has a non-trivial destructor, which means that (unlike a raw pointer) it cannot be passed in a CPU register when being passed to a function. This makes them slower than raw pointers.

Shared pointer

Shared ownership means that an object can have multiple owners. When the last owner ceases to exist, the object will be deleted. This is a very useful pointer type but is also more complicated than the unique pointer.

The `std::shared_ptr` object uses reference counting to keep track of the number of owners an object has. When the counter reaches 0, the object will be deleted. The counter needs to be stored somewhere, so it does have some memory overhead compared to the unique pointer. Also, `std::shared_ptr` is internally thread-safe, so the counter needs to be updated atomically to prevent race conditions.

The recommended way of creating objects owned by shared pointers is to use `std::make_shared<T>()`. It is both safer (from an exception-safety point of view) and more efficient than creating the object manually with `new` and then passing it to a `std::shared_ptr` constructor. By overloading `operator new()` and `operator delete()` again to track allocations, we can conduct an experiment to find out why using `std::make_shared<T>()` is more efficient:

```
auto operator new(size_t size) -> void* {
  void* p = std::malloc(size);
  std::cout << "allocated " << size << " byte(s)" << '\n';
  return p;
```

```
}
auto operator delete(void* p) noexcept -> void {
  std::cout << "deleted memory\n";
  return std::free(p);
}
```

Now, let's try the recommended way first, using `std::make_shared()`:

```
int main() {
  auto i = std::make_shared<double>(42.0);
  return 0;
}
```

The output when running the program is as follows:

```
allocated 32 bytes
deleted memory
```

Now, let's allocate the `int` value explicitly by using `new` and then pass it to the `std::shared_ptr` constructor:

```
int main() {
  auto i = std::shared_ptr<double>{new double{42.0}};
  return 0;
}
```

The program will generate the following output:

```
allocated 4 bytes
allocated 32 bytes
deleted memory
deleted memory
```

We can conclude that the second version needs two allocations, one for the `double` and one for the `std::shared_ptr`, whereas the first version only needed one allocation. This also means that, by using `std::make_shared()`, our code will be more cache-friendly, thanks to spatial locality.

Weak pointer

Weak ownership doesn't keep any objects alive; it only allows us to use an object if someone else owns it. Why would you want such a fuzzy ownership as weak ownership? One common reason for using a weak pointer is to break a reference cycle. A reference cycle occurs when two or more objects refer to each other using shared pointers. Even if all external `std::shared_ptr` constructors are gone, the objects are kept alive by referring to themselves.

Why not just use a raw pointer? Isn't the weak pointer exactly what a raw pointer already is? Not at all. A weak pointer is safe to use since we cannot reference the object unless it actually exists, which is not the case with a dangling raw pointer. An example will clarify this:

```
auto i = std::make_shared<int>(10);
auto weak_i = std::weak_ptr<int>{i};

// Maybe i.reset() happens here so that the int is deleted...
if (auto shared_i = weak_i.lock()) {
  // We managed to convert our weak pointer to a shared pointer
  std::cout << *shared_i << '\n';
}
```

```
else {
  std::cout << "weak_i has expired, shared_ptr was nullptr\n";
}
```

Whenever we try to use the weak pointer, we need to convert it into a shared pointer first using the member function `lock()`. If the object hasn't expired, the shared pointer will be a valid pointer to that object; otherwise, we will get an empty `std::shared_ptr` back. This way, we can avoid dangling pointers when using `std::weak_ptr` instead of raw pointers.

This will end our section about objects in memory. C++ offers excellent support for dealing with memory, both regarding low-level concepts such as alignment and padding and high-level concepts such as object ownership.

Having a sound understanding of ownership, RAII, and reference counting are very important when working with C++. Programmers that are new to C++ and haven't been exposed to these concepts earlier might need some time to fully grasp this. At the same time, these concepts are not unique to C++. In most languages, they are more diffused, but in others, they are even more prominent (Rust is an example of the latter). So, once mastered, it will improve your programming skills in other languages as well. Thinking about object ownership will have a positive impact of the design and architecture of the programs you write.

Now, we will move on to an optimization technique that will reduce the usage of dynamic memory allocations and instead use the stack whenever possible.

Small object optimization

One of the great things about containers such as `std::vector` is that they automatically allocate dynamic memory when needed. Sometimes, though, the use of dynamic memory for container objects that only

contain a few small elements can hurt performance. It would be more efficient to keep the elements in the container itself and only use stack memory, instead of allocating small regions of memory on the heap. Most modern implementations of `std::string` will take advantage of the fact that a lot of strings in a normal program are short, and that short strings are more efficient to handle without the use of heap memory.

One alternative is to keep a small separate buffer in the string class itself, which can be used when the string's content is short. This would increase the size of the string class, even when the short buffer is not used.

So, a more memory-efficient solution is to use a union, which can hold a short buffer when the string is in short mode and, otherwise, hold the data members it needs to handle a dynamically allocated buffer. The technique for optimizing a container for handling small data is usually referred to as small string optimization for strings, or small object optimization and small buffer optimization for other types. We have many names for the things we love.

A short code example will demonstrate how `std::string` from libc++ from LLVM behaves on my 64-bit system:

```
auto allocated = size_t{0};
// Overload operator new and delete to track allocations
void* operator new(size_t size) {
  void* p = std::malloc(size);
  allocated += size;
  return p;
}

void operator delete(void* p) noexcept {
  return std::free(p);
}
```

```
int main() {
  allocated = 0;
  auto s = std::string{""}; // Elaborate with different string sizes

  std::cout << "stack space = " << sizeof(s)
    << ", heap space = " << allocated
    << ", capacity = " << s.capacity() << '\n';
}
```

The code starts by overloading global `operator new` and `operator delete` for the purpose of tracking dynamic memory allocations. We can now start testing different sizes of the string `s` to see how `std::string` behaves. When building and running the preceding example in release mode on my system, it generates the following output:

```
stack space = 24, heap space = 0, capacity = 22
```

This output tells us that `std::string` occupies 24 bytes on the stack and that it has a capacity of 22 chars without using any heap memory. Let's verify that this is actually true by replacing the empty string with a string of 22 chars:

```
auto s = std::string{"12345678901234567890123"};
```

The program still produces the same output and verifies that no dynamic memory has been allocated. But what happens when we increase the string to hold 23 characters instead?

```
auto s = std::string{"123456789012345678901234"};
```

Running the program now produces the following output:

```
stack space = 24, heap space = 32, capacity = 31
```

The `std::string` class has now been forced to use the heap to store the string. It allocates 32 bytes and reports that the capacity is 31. This is because libc++ always stores a null-terminated string internally and, therefore, needs an extra byte at the end for the null character. It is still quite remarkable that the string class can be only 24 bytes and can hold strings of 22 characters in length without allocating any memory. How does it do this? As mentioned earlier, it is common to save memory by using a union with two different layouts: one for the short mode and one for the long mode. There is a lot of cleverness in the real libc++ implementation to make the maximum use of the 24 bytes that are available. The code here is simplified for the purpose of demonstrating this concept. The layout for the long mode looks like this:

```
struct Long {
  size_t capacity_{};
  size_t size_{};
  char* data_{};
};
```

Each member in the long layout is 8 bytes, so the total size is 24 bytes. The `char` pointer `data_` is a pointer to the dynamically allocated memory that will hold long strings. The layout of the short mode looks something like this:

```
struct Short {
  unsigned char size_{};
  char data_[23]{};
};
```

In the short mode, there is no need to use a variable for the capacity, since it is a compile-time constant. It is also possible to use a smaller type for the `size_` data member in this layout, since we know that the length of the string can only range from 0 to 22 if it is a short string.

Both layouts are combined using a union:

```
union u_ {
  Short short_layout_;
  Long long_layout_;
};
```

There is one piece missing, though: how can the string class know whether it is currently storing a short string or a long string? A flag is needed to indicate this, but where is it stored? It turns out that libc++ uses the least significant bit on the `capacity_` data member in the long mode, and the least significant bit on the `size_` data member in the short mode. For the long mode, this bit is redundant anyway since the string always allocates memory sizes that are multiples of 2. In the short mode, it is possible to use only 7 bits for storing the size so that one bit can be used for the flag. It becomes even more complicated when writing this code to handle big endian byte order, since the bit needs to be placed in memory at the same location, regardless of whether we are using the short struct or the long struct of the union. You can look up the details in the libc++ implementation at **https://github.com/llvm/llvm-project/tree/master/libcxx**.

Figure 7.9 summarizes our simplified (but still rather complicated) memory layout of the union used by an efficient implementation of the small string optimization:

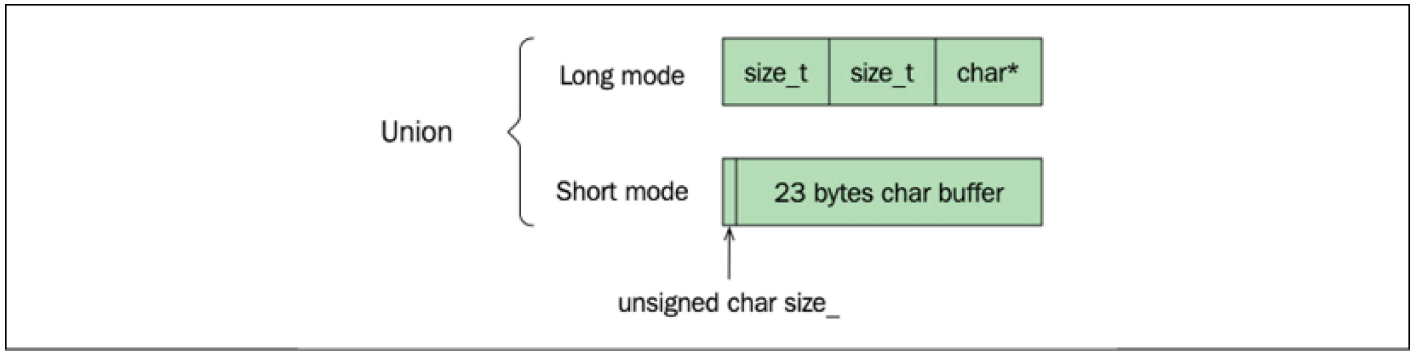

Figure 7.9: The union of the two different layouts used for handling short strings and long strings, respectively

Clever tricks like this are the reason that you should strive to use the efficient and well-tested classes provided by the standard library before you try to roll out your own. Nevertheless, knowing about these optimizations and how they work is important and useful, even if you never need to write one yourself.

Custom memory management

We have come a long way in this chapter now. We have covered the basics of virtual memory, the stack and the heap, the `new` and `delete` expressions, memory ownership, and alignment and padding. But before we close this chapter, we are going to show how to customize memory management in C++. We will see how the parts that we went through earlier in this chapter will come in handy when writing a custom memory allocator.

But first, what is a custom memory manager and why do we need one?

When using `new` or `malloc()` to allocate memory, we use the built-in memory management system in C++. Most implementations of `operator new` use `malloc()`, which is a general-purpose memory allocator. Designing and building a general-purpose memory manager is a complicated task, and there are

many people who have already spent a lot of time researching this topic. Still, there are several reasons why you might want to write a custom memory manager. Here are some examples:

- **Debugging and diagnostics**: We have already done this a couple of times in this chapter by overloading `operator new` and `operator delete`, just to print out some debugging information.
- **Sandboxing**: A custom memory manager can provide a sandbox for code that isn't allowed to allocate unrestricted memory. The sandbox can also track memory allocations and release memory when the sandboxed code finishes executing.
- **Performance**: If we need dynamic memory and can't avoid allocations, we may have to write a custom memory manager that performs better for our specific needs. Later on, we will cover some of the circumstances that we could utilize to outperform `malloc()`.

With that said, many experienced C++ programmers have never faced a problem that actually required them to customize the standard memory manager that comes with the system. This is a good indication of how good the general-purpose memory managers actually are today, despite all the requirements they have to fulfill without any knowledge about our specific use cases. The more we know about the memory usage patterns in our application, the better the chances are that we can actually write something more efficient than `malloc()`. Remember the stack, for example? Allocating and deallocating memory from the stack is very fast compared to the heap, thanks to the fact that it doesn't need to handle multiple threads and that deallocations are guaranteed to always happen in reverse order.

Building a custom memory manager usually starts with analyzing the exact memory usage patterns and then implementing an arena.

Building an arena

Two frequently used terms when working with memory allocators are **arena** and **memory pool**. We will not distinguish between these terms in this book. By arena, I mean a block of contiguous memory, including a strategy for handing out parts of that memory and reclaiming it later on.

An arena could technically also be called a *memory resource* or an *allocator*, but those terms will be used to refer to abstractions from the standard library. The custom allocator we will develop later will be implemented using the arena we create here.

There are some general strategies that can be used when designing an arena that will make allocations and deallocations likely to perform better than `malloc()` and `free()`:

- **Single-threaded**: If we know that an arena will only be used from one thread, there is no need to protect data with synchronization primitives, such as locks or atomics. There is no risk that the client using the arena may be blocked by some other thread, which is important in real-time contexts.
- **Fixed-size allocations**: If the arena only hands out memory blocks of a fixed size, it is relatively easy to reclaim memory efficiently without memory fragmentation by using a free list.
- **Limited lifetime**: If you know that objects allocated from an arena only need to live during a limited and well-defined lifetime, the arena can postpone reclamation and free the memory all at once. An example could be objects created while handling a request in a server application. When the request has finished, all the memory that was handed out during the request can be reclaimed in one step. Of course, the arena needs to be big enough to handle all the allocations during the request without reclaiming memory continually; otherwise, this strategy will not work.

I will not go into further details about these strategies, but it is good to be aware of the possibilities when looking for ways to improve memory management in your program. As is often the case with optimizing software, the key is to understand the circumstances under which your program will run and to analyze the specific memory usage patterns. We do this to find ways to improve a custom memory manager compared to a general-purpose one.

Next, we will have a look at a simple arena class template, which can be used for small or few objects that need dynamic storage duration, but where the memory it needs usually is so small that it can be placed on the stack. This code is based on Howard Hinnant's `short_alloc`, published at **https://howardhinnant.github.io/stack_alloc.html**. This is a great place to start if you want to dig deeper

into custom memory management. I think it is a good example for demonstration purposes because it can handle multiple sizes of objects, which require proper handling of alignment.

But again, keep in mind that this is a simplified version for demonstrating the concept rather than providing you with production-ready code:

```
template <size_t N>
class Arena {
  static constexpr size_t alignment = alignof(std::max_align_t);
public:
  Arena() noexcept : ptr_(buffer_) {}
  Arena(const Arena&) = delete;
  Arena& operator=(const Arena&) = delete;

  auto reset() noexcept { ptr_ = buffer_; }
  static constexpr auto size() noexcept { return N; }
  auto used() const noexcept {
    return static_cast<size_t>(ptr_ - buffer_);
  }
  auto allocate(size_t n) -> std::byte*;
  auto deallocate(std::byte* p, size_t n) noexcept -> void;

private:
  static auto align_up(size_t n) noexcept -> size_t {
    return (n + (alignment-1)) & ~(alignment-1);
  }
  auto pointer_in_buffer(const std::byte* p) const noexcept -> bool {
    return std::uintptr_t(p) >= std::uintptr_t(buffer_) &&
           std::uintptr_t(p) < std::uintptr_t(buffer_) + N;
  }
  alignas(alignment) std::byte buffer_[N];
```

```
  std::byte* ptr_{};
};
```

The arena contains an `std::byte` buffer, whose size is determined at compile time. This makes it possible to create an arena object on the stack or as a variable with a static or thread local storage duration. The alignment might be allocated on the stack; hence, there is no guarantee that it will be aligned for types other than `char` unless we apply the `alignas` specifier to the array. The helper `align_up()` function may look complicated if you are not used to bitwise operations. However, it basically just rounds up to the alignment requirement that we use. The memory that this version will hand out will be the same as when using `malloc()` as it's suitable for any type. This is a bit wasteful if we use the arena for small types with smaller alignment requirements, but we'll ignore this here.

When reclaiming memory, we need to know whether the pointer we are asked to reclaim actually belongs to our arena. The `pointer_in_buffer()` function checks this by comparing a pointer address with the address range of the arena. As a side note, relationally comparing raw pointers to disjoint objects is undefined behavior; this might be used by an optimizing compiler and result in surprising effects. To avoid this, we are casting the pointers to `std::uintptr_t` before comparing the addresses. If you are curious about the details behind this, you can find a thorough explanation in the article *How to check if a pointer is in range of memory* by Raymond Chen at **https://devblogs.microsoft.com/oldnewthing/20170927-00/?p=97095**.

Next, we need the implementation of allocate and deallocate:

```
template<size_t N>
auto Arena<N>::allocate(size_t n) -> std::byte* {
  const auto aligned_n = align_up(n);
  const auto available_bytes =
    static_cast<decltype(aligned_n)>(buffer_ + N - ptr_);
  if (available_bytes >= aligned_n) {
```

```
    auto* r = ptr_;
    ptr_ += aligned_n;
    return r;
  }
  return static_cast<std::byte*>(::operator new(n));
}
```

The `allocate()` function returns a pointer to a correctly aligned memory with the specified size, `n`. If there is no available space in the buffer for the requested size, it will fall back to using `operator new` instead.

The following `deallocate()` function first checks whether the pointer to the memory to be deallocated is from the buffer, or has been allocated with `operator new`. If it is not from the buffer, we simply delete it with `operator delete`. Otherwise, we check whether the memory to be deallocated is the last memory we handed out from the buffer and, then, reclaim it by moving the current `ptr_`, just as a stack would do. We simply ignore other attempts to reclaim the memory:

```
template<size_t N>
auto Arena<N>::deallocate(std::byte* p, size_t n) noexcept -> void {
  if (pointer_in_buffer(p)) {
    n = align_up(n);
    if (p + n == ptr_) {
      ptr_ = p;
    }
  }
  else {
    ::operator delete(p);
  }
}
```

That's about it; our arena is now ready to be used. Let's use it when allocating `User` objects:

```
auto user_arena = Arena<1024>{};

class User {
public:
  auto operator new(size_t size) -> void* {
    return user_arena.allocate(size);
  }
  auto operator delete(void* p) -> void {
    user_arena.deallocate(static_cast<std::byte*>(p), sizeof(User));
  }
  auto operator new[](size_t size) -> void* {
    return user_arena.allocate(size);
  }
  auto operator delete[](void* p, size_t size) -> void {
    user_arena.deallocate(static_cast<std::byte*>(p), size);
  }
private:
  int id_{};
};

int main() {
  // No dynamic memory is allocated when we create the users
  auto user1 = new User{};
  delete user1;

  auto users = new User[10];
  delete [] users;

  auto user2 = std::make_unique<User>();
```

```
  return 0;
}
```

The `User` objects created in this example will all reside in the buffer of the `user_area` object. That is, no dynamic memory is allocated when we call `new` or `make_unique()` here. But there are other ways to create `User` objects in C++ that this example doesn't show. We will cover them in the next section.

A custom memory allocator

When trying out our custom memory manager with a specific type, it worked great! There is a problem, though. It turns out that the class-specific `operator new` is not called on all the occasions that we might have expected. Consider the following code:

```
auto user = std::make_shared<User>();
```

What happens when we want to have a `std::vector` of 10 users?

```
auto users = std::vector<User>{};
users.reserve(10);
```

In neither of the two cases is our custom memory manager being used. Why? Starting with the shared pointer, we have to go back to the example earlier where we saw that `std::make_shared()` actually allocates memory for both reference counting data and the object it should point to. There is no way that `std::make_shared()` can use an expression such as `new User()` to create the user object and the counter with only one allocation. Instead, it allocates memory and constructs the user object using placement new.

The `std::vector` object is similar. It doesn't construct 10 objects by default in an array when we call `reserve()`. This would have required a default constructor for all the classes to be used with the vector. Instead, it allocates memory that can be used for holding 10 user objects when they are being added. Again, placement new is the tool for making this possible.

Fortunately, we can provide a custom memory allocator to both `std::vector` and `std::shared_ptr` in order to have them use our custom memory manager. This is true for the rest of the containers in the standard library as well. If we don't supply a custom allocator, the containers will use the default `std::allocator<T>` class. So, what we need in order to use our arena is to write an allocator that can be used by the containers.

Custom allocators have been a hotly debated topic for a long time in the C++ community. Many custom containers have been implemented to control how memory is managed instead of using the standard containers with custom allocators, probably for good reasons.

However, the support and requirements for writing a custom allocator were improved in C++11, and are now a lot better. Here, we will only focus on allocators from C++11 and beyond.

A minimal allocator in C++11 now looks like this:

```
template<typename T>
struct Alloc {
  using value_type = T;
  Alloc();
  template<typename U> Alloc(const Alloc<U>&);
  T* allocate(size_t n);
  auto deallocate(T*, size_t) const noexcept -> void;
};
template<typename T>
auto operator==(const Alloc<T>&, const Alloc<T>&) -> bool;
```

```
template<typename T>
auto operator!=(const Alloc<T>&, const Alloc<T>&) -> bool;
```

It's really not that much code anymore, thanks to the improvements in C++11. The container that uses the allocator actually uses `std::allocator_traits`, which provides reasonable defaults if the allocator omits them. I recommend you have a look at the `std::allocator_traits` to see what traits can be configured and what the defaults are.

By using `malloc()` and `free()`, we could quite easily implement a minimal custom allocator. Here, we will show the old and famous `Mallocator`, first published in a blog post by Stephan T. Lavavej, to demonstrate how to write a minimal custom allocator using `malloc()` and `free()`. Since then, it has been updated for C++11 to make it even slimmer. Here is how it looks:

```
template <class T>
struct Mallocator {

 using value_type = T;
 Mallocator() = default;

 template <class U>
 Mallocator(const Mallocator<U>&) noexcept {}

 template <class U>
 auto operator==(const Mallocator<U>&) const noexcept {
  return true;
 }

 template <class U>
 auto operator!=(const Mallocator<U>&) const noexcept {
  return false;
 }
```

```
  auto allocate(size_t n) const -> T* {
    if (n == 0) {
      return nullptr;
    }
    if (n > std::numeric_limits<size_t>::max() / sizeof(T)) {
      throw std::bad_array_new_length{};
    }
    void* const pv = malloc(n * sizeof(T));
    if (pv == nullptr) {
      throw std::bad_alloc{};
    }
    return static_cast<T*>(pv);
  }
  auto deallocate(T* p, size_t) const noexcept -> void {
    free(p);
  }
};
```

`Mallocator` is a **stateless allocator**, which means that the allocator instance itself doesn't have any mutable state; instead, it uses global functions for allocation and deallocation, namely `malloc()` and `free()`. A stateless allocator should always compare equal to the allocators of the same type. It indicates that memory allocated with `Mallocator` should also be deallocated with `Mallocator`, regardless of the `Mallocator` instance. A stateless allocator is the least complicated allocator to write, but it is also limited, since it depends on the global state.

To use our arena as a stack-allocated object, we will need a **stateful allocator** that can reference the arena instance. Here, the arena class that we implemented really starts to make sense. Say, for example, that we want to use one of the standard containers in a function to do some processing. We know that, most of the time, we are dealing with very small amounts of data that will fit on the stack. But once we

use the containers from the standard library, they will allocate memory from the heap, which, in this case, will hurt our performance.

What are the alternatives to using the stack to manage data and avoid unnecessary heap allocations? One alternative is to build a custom container that uses a variation of the small object optimization we looked at for `std::string`.

It is also possible to use a container from Boost, such as `boost::container::small_vector`, which is based on LLVM's small vector. We advise you to check it out if you haven't already: http://www.boost.org/doc/libs/1_74_0/doc/html/container/non_standard_containers.html.

Yet another alternative, though, is to use a custom allocator, which we will explore next. Since we already have an arena template class ready, we could simply create the instance of an arena on the stack and have a custom allocator use it for the allocations. What we then need to do is implement a stateful allocator, which could hold a reference to the stack-allocated arena object.

Again, this custom allocator that we will implement is a simplified version of Howard Hinnant's `short_alloc`:

```
template <class T, size_t N>
struct ShortAlloc {

  using value_type = T;
  using arena_type = Arena<N>;

  ShortAlloc(const ShortAlloc&) = default;
  ShortAlloc& operator=(const ShortAlloc&) = default;

  ShortAlloc(arena_type& arena) noexcept : arena_{&arena} { }

  template <class U>
```

```
  ShortAlloc(const ShortAlloc<U, N>& other) noexcept
    : arena_{other.arena_} {}

  template <class U> struct rebind {
   using other = ShortAlloc<U, N>;
  };
  auto allocate(size_t n) -> T* {
   return reinterpret_cast<T*>(arena_->allocate(n*sizeof(T)));
  }
  auto deallocate(T* p, size_t n) noexcept -> void {
   arena_->deallocate(reinterpret_cast<std::byte*>(p), n*sizeof(T));
  }
  template <class U, size_t M>
  auto operator==(const ShortAlloc<U, M>& other) const noexcept {
   return N == M && arena_ == other.arena_;
  }
  template <class U, size_t M>
  auto operator!=(const ShortAlloc<U, M>& other) const noexcept {
   return !(*this == other);
  }
  template <class U, size_t M> friend struct ShortAlloc;
 private:
  arena_type* arena_;
 };
```

The allocator holds a reference to the arena. This is the only state the allocator has. The functions `allocate()` and `deallocate()` simply forward their requests to the arena. The compare operators ensure that two instances of the `ShortAlloc` type are using the same arena.

Now, the allocator and arena we implemented can be used with a standard container to avoid dynamic memory allocations. When we are using small data, we can handle all allocations using the stack instead. Let's see an example using `std::set`:

```
int main() {

  using SmallSet =
    std::set<int, std::less<int>, ShortAlloc<int, 512>>;

  auto stack_arena = SmallSet::allocator_type::arena_type{};
  auto unique_numbers = SmallSet{stack_arena};

  // Read numbers from stdin
  auto n = int{};
  while (std::cin >> n)
    unique_numbers.insert(n);

  // Print unique numbers
  for (const auto& number : unique_numbers)
    std::cout << number << '\n';
}
```

The program reads integers from standard input until the end-of-file is reached (Ctrl + D on Unix-like systems and Ctrl + Z on Windows). It then prints the unique numbers in ascending order. Depending on how many numbers are read from `stdin`, the program will use stack memory or dynamic memory by using our `ShortAlloc` allocator.

Using polymorphic memory allocators

If you have followed this chapter, you now know how to implement a custom allocator that can be used with arbitrary containers, including those from the standard library. Suppose we want to use our new allocator for some code we find in our code base that is processing buffers of the type `std::vector<int>`, like this:

```
void process(std::vector<int>& buffer) {
  // ...
}
auto some_func() {
  auto vec = std::vector<int>(64);
  process(vec);
  // ...
}
```

We are eager to try out our new allocator, which is utilizing stack memory, and try to inject it like this:

```
using MyAlloc = ShortAlloc<int, 512>;  // Our custom allocator
auto some_func() {
  auto arena = MyAlloc::arena_type();
  auto vec = std::vector<int, MyAlloc>(64, arena);
  process(vec);
  // ...
}
```

When compiling, we come to the painful realization that `process()` is a function that expects `std::vector<int>`, and our `vec` variable is now of another type. GCC gives us the following error:

```
error: invalid initialization of reference of type 'const std::vector<int>&' from expression of type 'std::vector<int, ShortAlloc<int, 5
```

The reason for the type mismatch is that the custom allocator, `MyAlloc`, that we want to use is passed to `std::vector` as a template parameter and therefore becomes part of the type we instantiate. As a result, `std::vector<int>` and `std::vector<int, MyAlloc>` cannot be interchanged.

This may or may not be a problem for the use cases you are working on, and you could solve it by making the `process()` function accept a `std::span` or make it a generic function working with ranges instead of requiring a `std::vector`. Regardless, it's important to realize that the allocator actually becomes a part of the type when using allocator-aware template classes from the standard library.

What allocator is `std::vector<int>` using then? The answer is that `std::vector<int>` uses the default template argument which is `std::allocator`. So, writing `std::vector<int>` is equivalent to `std::vector<int, std::allocator<int>>`. The template class `std::allocator` is an empty class that uses global `new` and global `delete` when it fulfills allocation and deallocation requests from the container. This also means that the size of a container using an empty allocator is smaller than that of a container using our custom allocator:

```
std::cout << sizeof(std::vector<int>) << '\n';
// Possible output: 24
std::cout << sizeof(std::vector<int, MyAlloc>) << '\n';
// Possible output: 32
```

Checking the implementation of `std::vector` from libc++, we can see that it is using a nifty type called **compressed pair**, which, in turn, is based on the *empty base-class optimization* to get rid of the unnecessary storage usually occupied by a member of an empty class. We will not cover the details here, but if you are interested, you could have a look at the boost version of `compressed_pair`, which is documented at **https://www.boost.org/doc/libs/1_74_0/libs/utility/doc/html/compressed_pair.html**.

This problem of ending up with different types when using different allocators was addressed in C++17 by introducing an extra layer of indirection; all standard containers under the namespace `std::pmr` use the same allocator, namely `std::pmr::polymorphic_allocator`, which dispatches all allocation/deallocation requests to a **memory resource** class. So, instead of writing new custom memory allocators, we could use

the general polymorphic memory allocator named `std::pmr::polymorphic_allocator` and instead write new custom memory resources that will be handed to the polymorphic allocator during construction. The memory resource is analogous to our `Arena` class, and the `polymorphic_allocator` is the extra layer of indirection that contains a pointer to the resource.

The following diagram shows the control flow as the vector delegates to its allocator instance and the allocator, in turn, delegates to the memory resource to which it points:

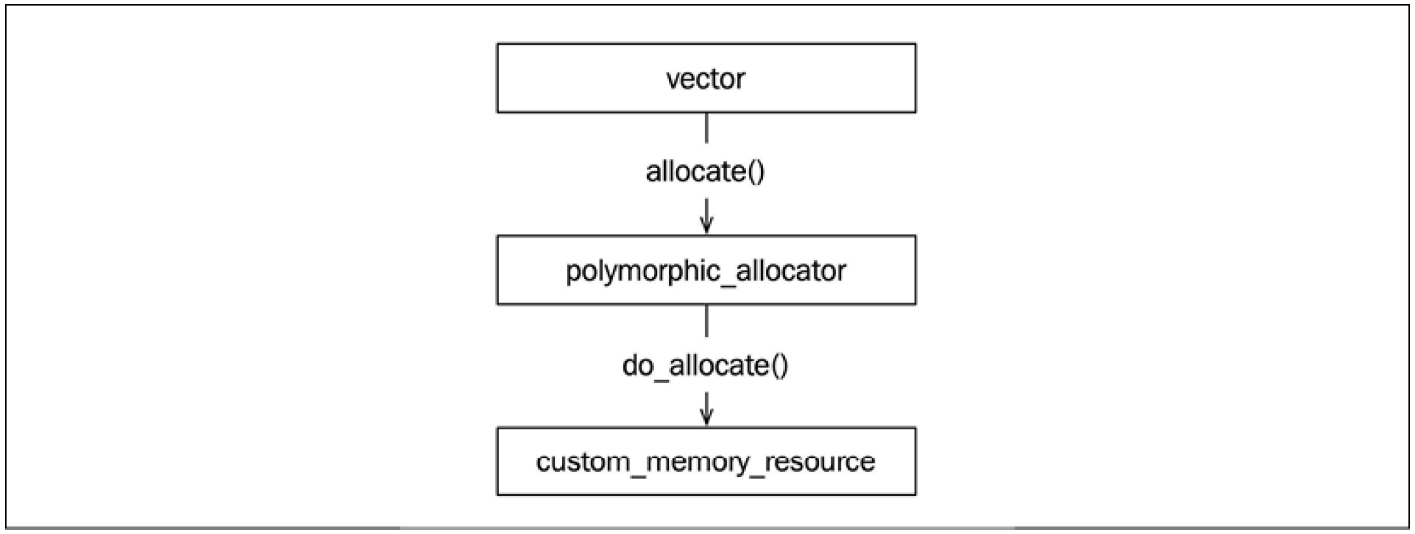

Figure 7.10: Allocating memory using a polymorphic_allocator

To start using the polymorphic allocator, we need to change the namespace from `std` to `std::pmr`:

```
auto v1 = std::vector<int>{};          // Uses std::allocator
auto v2 = std::pmr::vector<int>{/*...*/}; // Uses polymorphic_allocator
```

Writing a custom memory resource is relatively straightforward, especially with knowledge about memory allocators and arenas. But we might not even have to write a custom memory resource in order to achieve what we want. C++ already provides us with a few useful implementations that we should consider before writing our own. All memory resources derive from the base class `std::pmr::memory_resource`. The following memory resources live in the `<memory_resource>` header:

- `std::pmr::monotonic_buffer_resource`: This is quite similar to our `Arena` class. This class is preferable in scenarios when we're creating many objects with a short lifetime. Memory is freed only when the `monotonic_buffer_resource` instance is destructed, which makes allocations very fast.
- `std::pmr::unsynchronized_pool_resource`: This uses memory pools (also known as "slabs") containing fixed-size memory blocks, which avoids fragmentation within each pool. Each pool hands out memory for objects of a certain size. If you are creating many objects of a few different sizes, this class can be beneficial to use. This memory resource is not thread-safe and cannot be used from multiple threads unless you provide external synchronization.
- `std::pmr::synchronized_pool_resource`: This is a thread-safe version of `unsynchronized_pool_resource`.

Memory resources can be chained. When creating an instance of a memory resource, we can provide it with an **upstream memory resource**. This will be used if the current resource cannot handle the request (similar to what we did in `ShortAlloc` by using `malloc()` once our small buffer was full), or when the resource itself needs to allocate memory (such as when `monotonic_buffer_resource` needs to allocate its next buffer). The `<memory_resource>` header provides us with free functions that return pointers to global resource objects that are useful when specifying upstream resources:

- `std::pmr::new_delete_resource()`: Uses the global `operator new` and `operator delete`.
- `std::pmr::null_memory_resource()`: A resource that always throws `std::bad_alloc` whenever it is asked to allocate memory.
- `std::pmr::get_default_resource()`: Returns a globally default memory resource that can be set at runtime by `set_default_resource()`. The initial default resource is `new_delete_resource()`.

Let's see how we could rewrite our example from the last section, but this time using a `std::pmr::set`:

```
int main() {
  auto buffer = std::array<std::byte, 512>{};
  auto resource = std::pmr::monotonic_buffer_resource{
    buffer.data(), buffer.size(), std::pmr::new_delete_resource()};
  auto unique_numbers = std::pmr::set<int>{&resource};
  auto n = int{};
  while (std::cin >> n) {
    unique_numbers.insert(n);
  }
  for (const auto& number : unique_numbers) {
    std::cout << number << '\n';
  }
}
```

We are passing a stack-allocated buffer to the memory resource, and then providing it with the object returned from `new_delete_resource()` as an upstream resource to be used if the buffer becomes full. If we would have omitted the upstream resource, it would have used the default memory resource, which, in this case, would have been the same since our code does not change the default memory resource.

Implementing a custom memory resource

Implementing a custom memory resource is fairly simple. We need to publicly inherit from `std::pmr::memory_resource` and then implement three pure virtual functions that will be invoked by the base class (`std::pmr::memory_resource`). Let's implement a simple memory resource that prints allocations and deallocations and then forwards the request to the default memory resource:

```
class PrintingResource : public std::pmr::memory_resource {
public:
```

```
  PrintingResource() : res_{std::pmr::get_default_resource()} {}
private:
  void* do_allocate(std::size_t bytes, std::size_t alignment)override {
    std::cout << "allocate: " << bytes << '\n';
    return res_->allocate(bytes, alignment);
  }
  void do_deallocate(void* p, std::size_t bytes,
                     std::size_t alignment) override {
    std::cout << "deallocate: " << bytes << '\n';
    return res_->deallocate(p, bytes, alignment);
  }
  bool do_is_equal(const std::pmr::memory_resource& other)
    const noexcept override {
    return (this == &other);
  }
  std::pmr::memory_resource* res_;  // Default resource
};
```

Note that we are saving the default resource in the constructor rather than calling `get_default_resource()` directly from `do_allocate()` and `do_deallocate()`. The reason for this is that someone could potentially change the default resource by calling `set_default_resource()` in the time between an allocation and a deallocation.

We can use a custom memory resource to track allocations made by a `std::pmr` container. Here is an example of using a `std::pmr::vector`:

```
auto res = PrintingResource{};
auto vec = std::pmr::vector<int>{&res};
vec.emplace_back(1);
vec.emplace_back(2);
```

A possible output when running the program is:

```
allocate: 4
allocate: 8
deallocate: 4
deallocate: 8
```

Something to be very careful about when using polymorphic allocators is that we are passing around raw non-owning pointers to memory resources. This is not specific to polymorphic allocators; we actually had the same problem with our `Arena` class and `ShortAlloc` as well, but this might be even easier to forget when using containers from `std::pmr` since these containers are using the same allocator type. Consider the following example:

```
auto create_vec() -> std::pmr::vector<int> {
  auto resource = PrintingResource{};
  auto vec = std::pmr::vector<int>{&resource}; // Raw pointer
  return vec;                                  // Ops! resource
}                                              // destroyed here
auto vec = create_vec();
vec.emplace_back(1);                           // Undefined behavior
```

Since the resource is destroyed when it goes out if scope at the end of `create_vec()`, our newly created `std::pmr::vector` is useless and will most likely crash when used.

This concludes our section on custom memory management. It is a complicated subject and if you feel tempted to use custom memory allocators to gain performance, I encourage you to carefully measure and analyze the memory access patterns in your application before you use and/or implement custom allocators. Typically, there are only a small set of classes or objects in an application that really need to be

tweaked using custom allocators. At the same time, reducing the number of dynamic memory allocations in an application or grouping objects together, in certain regions of memory, can have a dramatic effect on performance.

Summary

This chapter has covered a lot of ground, starting with the basics of virtual memory and finally implementing a custom allocator that can be used by containers from the standard library. A good understanding of how your program uses memory is important. Overuse of dynamic memory can be a performance bottleneck that you might need to optimize away.

Before you start implementing your own containers or custom memory allocators, bear in mind that many people before you have probably had very similar memory issues to the ones you may face. So, there is a good chance that the right tool for you is already out there in a library. Building custom memory managers that are fast, safe, and robust is a challenge.

In the next chapter, you will learn how to benefit from the newly introduced feature of C++ concepts, and how we can use template metaprogramming to have the compiler generate code for us.