```
// > g++ -std=c++17 scratchpad.cpp; ./a.out
// delete myclass2
// delete myclass1
```

As shown in this example, even though a cyclic reference still exists, the allocated pointers are released back to the heap when `main()` completes. Finally, it should be noted that it is possible to convert `std::unique_ptr` to `std::shared_ptr` using the following syntax:

```
auto ptr = std::make_unique<int>();
std::shared_ptr<int> shared = std::move(ptr);
```

Since `std::unique_ptr` is being moved, it no longer owns the pointer, and instead `std::shared_ptr` now owns the pointer. Moving from `std::shared_ptr` to `std::unqiue_ptr` is not allowed.

Learning about mapping and permissions

In this section, the reader will learn how to map memory using C++ patterns. You will learn how to map memory (a common system-programming technique), while doing so using C++ patterns.

The basics

`malloc()`/`free()`, `new()`/`delete()`, and `std::unique_ptr{}`/`std::shared_ptr{}` are not the only methods for allocating memory on a POSIX system. C++-style allocators are another, more complicated, method for allocating memory that will be discussed in greater detail in `Chapter 9`, *A Hands-On Approach to Allocators*. A more direct, POSIX style for allocating memory is to use `mmap()`:

```
#include <iostream>
#include <sys/mman.h>

constexpr auto PROT_RW = PROT_READ | PROT_WRITE;
constexpr auto MAP_ALLOC = MAP_PRIVATE | MAP_ANONYMOUS;

int main()
{
    auto ptr = mmap(0, 0x1000, PROT_RW, MAP_ALLOC, -1, 0);
    std::cout << ptr << '\n';

    munmap(ptr, 0x1000);
}
```

```
// > g++ -std=c++17 scratchpad.cpp; ./a.out
// 0x7feb41ab6000
```

The `mmap()` function may be used to map memory from different sources into a program. For example, if you want to make device memory into your application, you would use `mmap()`. If `MAP_ANONYMOUS` is passed to `mmap()`, it can be used to allocate memory the same way you would allocate memory using `malloc()` and `free()`. In the preceding example, `mmap()` is used to allocate a 4k page of memory that is marked read/write. The use of `MAP_PRIVATE` tells `mmap()` that you do not intend to share this memory with other applications (for example, for interprocess communication). Mapping memory this way compared to `malloc()`/`free()` has some advantages and disadvantages.

Advantages:

- **Fragmentation**: Allocating memory using `MAP_ANONYMOUS` usually maps memory in sizes that are multiples of a page size, or, worst case, a power of two. The is because `mmap()` is asking the OS kernel for a block memory, and that memory must be mapped into the application, which can only be done in blocks no smaller than a page. As a result, fragmentation of this memory is far less likely that multiple, random memory allocations usually made using `malloc()`.
- **Permissions**: When using `mmap()`, you can state the permissions you wish to apply to the newly-allocated memory. This is especially useful if you need memory with special permissions, such as read/execute memory.
- **Shared memory**: The memory allocated using `mmap()` can also be shared by another application instead of being allocated privately for a specific application, as with `malloc()`.

Disadvantages:

- **Performance**: `malloc()`/`free()` allocate and deallocate to a block of memory that is managed by the C library inside the application itself. If more memory is needed, the C library will call into the OS, using functions such as `brk()` or even `mmap()`, to get more memory from the OS. When free is called, the released memory is provided back to the memory being managed by the C library, and in a lot of cases is never actually provided back to the OS. For this reason, `malloc()`/`free()` can quickly allocate memory for the application because no OS-specific calls are being made (unless of course the C library runs out of memory). `mmap()`, on the other hand, has to call into the OS on every single allocation. For this reason, it does not perform as well as `malloc()`/`free()` since an OS call can be expensive.

- **Granularity**: For the same reason that `mmap()` reduces fragmentation, it also reduces granularity. Every single allocation made by `mmap()` is at least a page in size, even if the requested memory is only a byte.

To demonstrate the potential waste of `mmap()`, see the following:

```
#include <iostream>
#include <sys/mman.h>

constexpr auto PROT_RW = PROT_READ | PROT_WRITE;
constexpr auto MAP_ALLOC = MAP_PRIVATE | MAP_ANONYMOUS;

int main()
{
    auto ptr1 = mmap(0, 42, PROT_RW, MAP_ALLOC, -1, 0);
    auto ptr2 = mmap(0, 42, PROT_RW, MAP_ALLOC, -1, 0);

    std::cout << ptr1 << '\n';
    std::cout << ptr2 << '\n';

    munmap(ptr1, 42);
    munmap(ptr2, 42);
}

// > g++ -std=c++17 scratchpad.cpp; ./a.out
// 0x7fc1637ad000
// 0x7fc1637ac000
```

In this example, 42 bytes are allocated twice, but the resulting addresses are a 4k page apart. This is because allocations made by `mmap()` must be at least a page in size, even though the requested amount was only 42 bytes. The reason that `malloc()`/`free()` does not have this waste is that these functions request large chunks of memory at a time from the OS, and then manage this memory using various different allocation schemes internally within the C library. For more information on how this is done, there is a very good explanation within `newlib` on the topic: `https://sourceware.org/git/?p=newlib-cygwin.git;a=blob;f=newlib/libc/stdlib/malloc.c`.

Permissions

`mmap()` may be used to allocate memory with special parameters. For example, suppose you need to allocate memory that has read/execute permissions instead of the read/write permissions that are typically associated with `malloc()`/`free()`:

```
#include <iostream>
#include <sys/mman.h>

constexpr auto PROT_RE = PROT_READ | PROT_EXEC;
constexpr auto MAP_ALLOC = MAP_PRIVATE | MAP_ANONYMOUS;

int main()
{
    auto ptr = mmap(0, 0x1000, PROT_RE, MAP_ALLOC, -1, 0);
    std::cout << ptr << '\n';

    munmap(ptr, 0x1000);
}

// > g++ -std=c++17 scratchpad.cpp; ./a.out
// 0x7feb41ab6000
```

As shown, allocating memory with read/execute permissions is the same as allocating memory with read/write permissions substituting `PROT_WRITE` with `PROT_EXEC`.

On systems that support read/write or read/execute (also known as W^E, which states that write is mutually exclusive with execute), write and execute permissions should not be used together at the same time. Specifically, in the event of malicious use of your program, preventing executable memory from also having write permissions can prevent a number of known cyber attacks.

The problem with allocating memory as read/execute and not read/write/execute is that there is no easy way to place executable code into your newly-allocated buffer as the memory was marked as read/execute only. The same is true if you wish to allocate read-only memory. Once again, since write permissions were never added, there is no way to add data to read-only memory as it doesn't have write permissions.

To make the situation worse, some operating systems prevent applications from allocating read/write/execute memory as they attempt to enforce W^E permissions. To overcome this issue, while still providing a means to set the desired permissions, POSIX provides `mprotect()`, which allows you to change the permissions of memory that has already been allocated. Although this may be used with memory that is managed by `malloc()`/`free()`, it should instead be used with `mmap()` memory permissions that can only be enforced at the page level on most architectures. `malloc()`/`free()` allocate from a large buffer that is shared among all of the program's allocations, while `mmap()` only allocates memory with page granularity, and therefore is not shared by other allocations.

The following shows an example of how to use `mprotect`:

```
#include <iostream>
#include <sys/mman.h>

constexpr auto PROT_RW = PROT_READ | PROT_WRITE;
constexpr auto MAP_ALLOC = MAP_PRIVATE | MAP_ANONYMOUS;

int main()
{
    auto ptr = mmap(0, 0x1000, PROT_RW, MAP_ALLOC, -1, 0);
    std::cout << ptr << '\n';

    if (mprotect(ptr, 0x1000, PROT_READ) == -1) {
        std::clog << "ERROR: Failed to change memory permissions\n";
        ::exit(EXIT_FAILURE);
    }

    munmap(ptr, 0x1000);
}

// > g++ -std=c++17 scratchpad.cpp; ./a.out
// 0x7fb05b4b6000
```

In this example, `mmap()` is used to allocate a buffer the size of a 4k page with read/write permissions. Once the memory is allocated, `mprotect()` is used to change the permissions of the memory to read-only. Finally, `munmap()` is used to release the memory back to the operating system.

Smart pointers and mmap()

With respect to C++, the biggest issue with `mmap()` and `munmap()` is that they suffer from a lot of the same disadvantages as `malloc()`/`free()`:

- **Memory leaks**: Since `mmap()` and `munmap()` must be executed manually, it's possible the user could forget to call `munmap()` when the memory is no longer needed, or a complex logic bug could result in `munmap()` not being called at the right time.
- **Memory mismatch**: It's possible that the users of `mmap()` could call `free()` instead of `munmap()` by accident, resulting in a mismatch that is almost certain to generate instability because memory from `mmap()` is coming from the OS kernel, while `free()` is expecting memory from application heap.

To overcome this, `mmap()` should be wrapped with `std::unique_ptr{}`:

```
#include <memory>
#include <iostream>

#include <string.h>
#include <sys/mman.h>

constexpr auto PROT_RW = PROT_READ | PROT_WRITE;
constexpr auto MAP_ALLOC = MAP_PRIVATE | MAP_ANONYMOUS;

class mmap_deleter
{
    std::size_t m_size;

public:
    mmap_deleter(std::size_t size) :
        m_size{size}
    { }

    void operator()(int *ptr) const
    {
        munmap(ptr, m_size);
    }
};

template<typename T, typename... Args>
auto mmap_unique(Args&&... args)
{
    if (auto ptr = mmap(0, sizeof(T), PROT_RW, MAP_ALLOC, -1, 0)) {

        auto obj = new (ptr) T(args...);
```

```
            auto del = mmap_deleter(sizeof(T));

            return std::unique_ptr<T, mmap_deleter>(obj, del);
        }

        throw std::bad_alloc();
    }

    int main()
    {
        auto ptr = mmap_unique<int>(42);
        std::cout << *ptr << '\n';
    }

    // > g++ -std=c++17 scratchpad.cpp; ./a.out
    // 42
```

In this example, the main function calls `mmap_unique()` instead of `std::make_unqiue()`, as `std::make_unique()` allocates memory using `new()`/`delete()`, and we wish to use `mmap()`/`munmap()` instead. The first part of the `mmap_unique()` function allocates memory using `mmap()` the same way as our previous examples. In this case, permissions were set to read/write, but they could have also been changed using `mprotect()` to provide read-only or read/execute if desired. If the call to `mmap()` fails, `std::bad_alloc()` is thrown, just like the C++ library.

The next line in this example uses the `new()` placement operator, as discussed earlier in in the *Placement new* section. The goal of this call is to create an object whose constructor has been called to initialize the `T` type as required. In the case of this example, this is setting an integer to `42`, but if a class were used instead of an integer, the classes constructor would be called with whatever arguments were passed to `mmap_unique()`.

The next step is to create a custom deleter for our `std::unqiue_ptr{}`. This is done because by default, `std::unqiue_ptr{}` will call the `delete()` operator instead of `munmap()`. The custom deleter takes a single argument that is the size of the original allocation. This is needed because `munmap()` needs to know the size of the original allocation, unlike `delete()` and `free()`, which just take a pointer.

Finally, `std::unique_ptr{}` is created with the newly-created object and custom deleter. From this point on, all of the memory that was allocated using `mmap()` can be accessed using the standard `std::unique_ptr{}` interface, and treated as a normal allocation. When the pointer is no longer needed, and `std::unique_ptr{}` is out of scope, the pointer will be released back to the OS kernel by calling `munmap()` as expected.

Shared memory

In addition to allocating memory, `mmap()` may be used to allocate shared memory, typically for interprocess communications. To demonstrate this, we start by defining a shared memory name, `"/shm"`, and our read, write, and execute permissions:

```
#include <memory>
#include <iostream>

#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <sys/mman.h>

constexpr auto PROT_RW = PROT_READ | PROT_WRITE;

auto name = "/shm";
```

Next, we must define our custom deleter, which uses `munmap()` instead of `free()`:

```
class mmap_deleter
{
    std::size_t m_size;

public:
    mmap_deleter(std::size_t size) :
        m_size{size}
    { }

    void operator()(int *ptr) const
    {
        munmap(ptr, m_size);
    }
};
```

In this example, we build off of the previous example, but instead of having a single `mmap_unique()` function, we now have a server and a client version. Although typically shared memory would be used for interprocess communication, in this example, we share memory in the same application to keep things simple.

The main function creates both a server and a client-shared pointer. The server version creates shared memory using the following:

```
template<typename T, typename... Args>
auto mmap_unique_server(Args&&... args)
{
  if(int fd = shm_open(name, O_CREAT | O_RDWR, 0644); fd != -1) {
      ftruncate(fd, sizeof(T));

        if (auto ptr = mmap(0, sizeof(T), PROT_RW, MAP_SHARED, fd, 0)) {

            auto obj = new (ptr) T(args...);
            auto del = mmap_deleter(sizeof(T));

            return std::unique_ptr<T, mmap_deleter>(obj, del);
        }
    }

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

This function is similar to the mmap_unique() function in the previous example, but opens a handle to a shared memory file instead of allocating memory using MAP_ANONYMOUS. To open the shared memory file, we use the POSIX shm_open() function. This function is similar to the open() function. The first parameter is the name of the shared memory file. The second parameter defines how the file is opened, while the third parameter provides the mode. shm_open() is used to open the shared memory file, and the file descriptor is checked to make sure the allocation succeeded (that is, the file descriptor is not -1).

Next, the file descriptor is truncated. This ensures that the size of the shared memory file is equal to the size of the memory we wish to share. In this case, we wish to share a single T type, so we need to get the size of T. Once the shared memory file has been properly sized, we need to map in the shared memory using mmap(). The call to mmap() is the same as our previous examples, with the exception that MAP_SHARED is used.

Finally, like the previous example, we leverage the new() placement operator to create the newly-allocated type in shared memory, we create the custom deleter, and then finally, we return std::unique_ptr{} for this shared memory.

To connect to this shared memory (which could be done from another application), we need to use the client version of the mmap_unique() function:

```
template<typename T>
auto mmap_unique_client()
{
  if(int fd = shm_open(name, O_RDWR, 0644); fd != -1) {
      ftruncate(fd, sizeof(T));

        if (auto ptr = mmap(0, sizeof(T), PROT_RW, MAP_SHARED, fd, 0)) {

            auto obj = static_cast<T*>(ptr);
            auto del = mmap_deleter(sizeof(T));

            return std::unique_ptr<T, mmap_deleter>(obj, del);
        }
    }

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

The server and client versions of these functions look similar, but there are differences. First and foremost, the shared memory file is opened without `O_CREAT`. This is because the server creates the shared memory file, while the client connects to the shared memory file, so there is no need to pass `O_CREAT` in the client version. Finally, the signature of the client version of this function doesn't take any arguments like the server version. This is because the server version uses the `new()` placement to initialize the shared memory, which doesn't need to be done a second time. Instead of using the new placement, `static_cast()` is used to convert `void *` to the proper type prior to delivering the pointer to the newly created `std::unique_ptr{}`:

```
int main()
{
    auto ptr1 = mmap_unique_server<int>(42);
    auto ptr2 = mmap_unique_client<int>();
    std::cout << *ptr1 << '\n';
    std::cout << *ptr2 << '\n';
}

// > g++ -std=c++17 scratchpad.cpp -lrt; ./a.out
// 42
// 42
```

The result of this example is that memory is shared between a server and a client, wrapping the shared memory in `std::unique_ptr{}`. Furthermore, as shown in the example, the memory is properly shared, as can be seen by `42` being printed for both the server and client version of the pointer. Although we use this for an integer type, this type of shared memory can be used with any complex type as needed (although care should be taken when attempting to share classes, especially those that leverage inheritance and contain `vTable`).