Non-transferability

You cannot move or copy a scoped_ptr, making it non-transferable. Listing 11-8 illustrates how attempting to move or copy a scoped_ptr results in an invalid program.

```
void by_ref(const ScopedOathbreakers&) { } ❶
void by_val(ScopedOathbreakers) { } ❷

TEST_CASE("ScopedPtr can") {
  ScopedOathbreakers aragorn{ new DeadMenOfDunharrow };
  SECTION("be passed by reference") {
    by_ref(aragorn); ❸
  }
  SECTION("not be copied") {
    // DOES NOT COMPILE:
    by_val(aragorn); ❹
    auto son_of_arathorn = aragorn; ❺
  }
  SECTION("not be moved") {
    // DOES NOT COMPILE:
    by_val(std::move(aragorn)); ❻
    auto son_of_arathorn = std::move(aragorn); ❼
  }
}
```

Listing 11-8: The boost::scoped_ptr is non-transferable. (This code doesn't compile.)

First, you declare dummy functions that take a scoped_ptr by reference ❶ and by value ❷. You can still pass a scoped_ptr by reference ❸, but attempting to pass one by value will fail to compile ❹. Also, attempting to use the scoped_ptr copy constructor or a copy assignment operator ❺ will fail to compile. In addition, if you try to move a scoped_ptr with std::move, your code won't compile ❻❼.

NOTE *Generally, using a boost::scoped_ptr incurs no overhead compared with using a raw pointer.*

boost::scoped_array

The boost::scoped_array is a scoped pointer for dynamic arrays. It supports the same usages as a boost::scoped_ptr, but it also implements an operator[] so you can interact with elements of the scoped array in the same way as you can with a raw array. Listing 11-9 illustrates this additional feature.

```
TEST_CASE("ScopedArray supports operator[]") {
  boost::scoped_array<int❶> squares{
    new int❷[5] { 0, 4, 9, 16, 25 }
  };
  squares[0] = 1; ❸
  REQUIRE(squares[0] == 1); ❹
```

```
  REQUIRE(squares[1] == 4);
  REQUIRE(squares[2] == 9);
}
```

Listing 11-9: The `boost::scoped_array` *implements* `operator[]`*.*

You declare a `scoped_array` the same way you declare a `scoped_ptr`, by using a single template parameter ❶. In the case of `scoped_array`, the template parameter is the type contained by the array ❷, not the type of the array. You pass in a dynamic array to the constructor of `squares`, making the dynamic array `squares` the array's owner. You can use `operator[]` to write ❸ and read ❹ elements.

A Partial List of Supported Operations

So far, you've learned about the major features of scoped pointers. For reference, Table 11-1 enumerates all the operators discussed, plus a few that haven't been covered yet. In the table, `ptr` is a raw pointer and `s_ptr` is a scoped pointer. See the Boost documentation for more information.

Table 11-1: All of the Supported `boost::scoped_ptr` Operations

Operation	Notes
`scoped_ptr<...>{ }` or `scoped_ptr <...>{ nullptr }`	Creates an empty scoped pointer.
`scoped_ptr <...>{ ptr }`	Creates a scoped pointer owning the dynamic object pointed to by **`ptr`**.
`~scoped_ptr<...>()`	Calls **`delete`** on the owned object if full.
`s_ptr1.swap(s_ptr2)`	Exchanges owned objects between **`s_ptr1`** and **`s_ptr2`**.
`swap(s_ptr1, s_ptr2)`	A free function identical to the `swap` method.
`s_ptr.reset()`	If full, calls `delete` on object owned by **`s_ptr`**.
`s_ptr.reset(ptr)`	Deletes currently owned object and then takes ownership of **`ptr`**.
`ptr = s_ptr.get()`	Returns the raw pointer **`ptr`**; **`s_ptr`** retains ownership.
`*s_ptr`	Dereferences operator on owned object.
`s_ptr->`	Member dereferences operator on owned object.
`bool{ s_ptr }`	`bool` conversion: `true` if full, `false` if empty.

Unique Pointers

A *unique pointer* has transferable, exclusive ownership over a single dynamic object. You *can* move unique pointers, which makes them transferable. They also have exclusive ownership, so they *cannot* be copied. The stdlib has a `unique_ptr` available in the `<memory>` header.

NOTE *Boost doesn't offer a unique pointer.*

Constructing

The `std::unique_ptr` takes a single template parameter corresponding to the pointed-to type, as in `std::unique_ptr<int>` for a "unique pointer to `int`" type.

As with a scoped pointer, the unique pointer has a default constructor that initializes the unique pointer to empty. It also provides a constructor taking a raw pointer that takes ownership of the pointed-to dynamic object. One construction method is to create a dynamic object with `new` and pass the result to the constructor, like this:

```
std::unique_ptr<int> my_ptr{ new int{ 808 } };
```

Another method is to use the `std::make_unique` function. The `make_unique` function is a template that takes all the arguments and forwards them to the appropriate constructor of the template parameter. This obviates the need for `new`. Using `std::make_unique`, you could rewrite the preceding object initialization as:

```
auto my_ptr = make_unique<int>(808);
```

The `make_unique` function was created to avoid some devilishly subtle memory leaks that used to occur when you used `new` with previous versions of C++. However, in the latest version of C++, these memory leaks no longer occur. Which constructor you use mainly depends on your preference.

Supported Operations

The `std::unique_ptr` function supports every operation that `boost::scoped_ptr` supports. For example, you can use the following type alias as a drop-in replacement for `ScopedOathbreakers` in Listings 11-1 to 11-7:

```
using UniqueOathbreakers = std::unique_ptr<DeadMenOfDunharrow>;
```

One of the major differences between unique and scoped pointers is that you can move unique pointers because they're *transferable*.

Transferable, Exclusive Ownership

Not only are unique pointers transferable, but they have exclusive ownership (you *cannot* copy them). Listing 11-10 illustrates how you can use the move semantics of `unique_ptr`.

```
TEST_CASE("UniquePtr can be used in move") {
  auto aragorn = std::make_unique<DeadMenOfDunharrow>(); ❶
  SECTION("construction") {
    auto son_of_arathorn{ std::move(aragorn) }; ❷
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❸
  }
  SECTION("assignment") {
    auto son_of_arathorn = std::make_unique<DeadMenOfDunharrow>(); ❹
```

```
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 2); ❺
    son_of_arathorn = std::move(aragorn); ❻
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❼
  }
}
```

Listing 11-10: The `std::unique_ptr` supports move semantics for transferring ownership.

This listing creates a `unique_ptr` called `aragorn` ❶ that you use in two separate tests.

In the first test, you move `aragorn` with `std::move` into the move constructor of `son_of_arathorn` ❷. Because `aragorn` transfers ownership of its `DeadMenOfDunharrow` to `son_of_arathorn`, the `oaths_to_fulfill` object still only has value 1 ❸.

The second test constructs `son_of_arathorn` via `make_unique` ❹, which pushes the `oaths_to_fulfill` to 2 ❺. Next, you use the move assignment operator to move `aragorn` into `son_of_arathorn` ❻. Again, `aragorn` transfers ownership to `son_of_aragorn`. Because `son_of_aragorn` can own only one dynamic object at a time, the move assignment operator destroys the currently owned object before emptying the dynamic object of `aragorn`. This results in `oaths_to_fulfill` decrementing to 1 ❼.

Unique Arrays

Unlike `boost::scoped_ptr`, `std::unique_ptr` has built-in dynamic array support. You just use the array type as the template parameter in the unique pointer's type, as in `std::unique_ptr<int[]>`.

It's *very important* that you don't initialize a `std::unique_ptr<T>` with a dynamic array `T[]`. Doing so will cause undefined behavior, because you'll be causing a `delete` of an array (rather than `delete[]`). The compiler cannot save you, because `operator new[]` returns a pointer that is indistinguishable from the kind returned by `operator new`.

Like `scoped_array`, a `unique_ptr` to array type offers `operator[]` for accessing elements. Listing 11-11 demonstrates this concept.

```
TEST_CASE("UniquePtr to array supports operator[]") {
  std::unique_ptr<int[]❶> squares{
    new int[5]{ 1, 4, 9, 16, 25 } ❷
  };
  squares[0] = 1; ❸
  REQUIRE(squares[0] == 1); ❹
  REQUIRE(squares[1] == 4);
  REQUIRE(squares[2] == 9);
}
```

Listing 11-11: The `std::unique_ptr` to an array type supports `operator[]`.

The template parameter `int[]` ❶ indicates to `std::unique_ptr` that it owns a dynamic array. You pass in a newly minted dynamic array ❷ and then use `operator[]` to set the first element ❸; then you use `operator[]` to retrieve elements ❹.

Deleters

The `std::unique_ptr` has a second, optional template parameter called its deleter type. A unique pointer's *deleter* is what gets called when the unique pointer needs to destroy its owned object.

A `unique_ptr` instantiation contains the following template parameters:

```
std::unique_ptr<T, Deleter=std::default_delete<T>>
```

The two template parameters are `T`, the type of the owned dynamic object, and `Deleter`, the type of the object responsible for freeing an owned object. By default, `Deleter` is `std::default_delete<T>`, which calls `delete` or `delete[]` on the dynamic object.

To write a custom deleter, all you need is a function-like object that is invokable with a `T*`. (The unique pointer will ignore the deleter's return value.) You pass this deleter as the second parameter to the unique pointer's constructor, as shown in Listing 11-12.

```
#include <cstdio>

auto my_deleter = [](int* x) { ❶
  printf("Deleting an int at %p.", x);
  delete x;
};
std::unique_ptr<int❷, decltype(my_deleter)❸> my_up{
  new int,
  my_deleter
};
```

Listing 11-12: Passing a custom deleter to a unique pointer

The owned object type is `int` ❷, so you declare a `my_deleter` function object that takes an `int*` ❶. You use `decltype` to set the deleter template parameter ❸.

Custom Deleters and System Programming

You use a custom deleter whenever `delete` doesn't provide the resource-releasing behavior you require. In some settings, you'll never need a custom deleter. In others, like system programming, you might find them quite useful. Consider a simple example where you manage a file using the low-level APIs `fopen`, `fprintf`, and `fclose` in the `<cstdio>` header.

The `fopen` function opens a file and has the following signature:

```
FILE*❶ fopen(const char *filename❷, const char *mode❸);
```

On success, `fopen` returns a non-`nullptr`-valued `FILE*` ❶. On failure, `fopen` returns `nullptr` and it sets the static `int` variable `errno` equal to an error code, like access denied (`EACCES = 13`) or no such file (`ENOENT = 2`).

NOTE *See the* `errno.h` *header for a listing of all error conditions and their corresponding* `int` *values.*

The `FILE*` file handle is a reference to a file the operating system manages. A *handle* is an opaque, abstract reference to some resource in an operating system. The `fopen` function takes two arguments: `filename` ❷ is the path to the file you want to open, and `mode` ❸ is one of the six options shown in Table 11-2.

Table 11-2: All Six `mode` Options for `fopen`

String	Operations	File exists:	File doesn't exist:	Notes
`r`	Read		`fopen` fails	
`w`	Write	Overwrite	Create it	If the file exists, all contents are discarded.
`a`	Append		Create it	Always write to the end of the file.
`r+`	Read/Write		`fopen` fails	
`w+`	Read/Write	Overwrite	Create it	If the file exists, all contents are discarded.
`a+`	Read/Write		Create it	Always write to the end of the file.

You must close the file manually with `fclose` once you're done using it. Failure to close file handles is a common source of resource leakages, like so:

```
void fclose(FILE* file);
```

To write to a file, you can use the `fprintf` function, which is like a `printf` that prints to a file instead of the console. The `fprintf` function has identical usage to `printf` except you provide a file handle as the first argument before the format string:

```
int❶ fprintf(FILE* file❷, const char* format_string❸, ...❹);
```

On success, `fprintf` returns the number of characters ❶ written to the open file ❷. The `format_string` is the same as the format string for `printf` ❸, as are the variadic arguments ❹.

You can use a `std::unique_ptr` to a `FILE`. Obviously, you don't want to call `delete` on the `FILE*` file handle when you're ready to close the file. Instead, you need to close with `fclose`. Because `fclose` is a function-like object accepting a `FILE*`, it's a suitable deleter.

The program in Listing 11-13 writes the string `HELLO, DAVE.` to the file `HAL9000` and uses a unique pointer to perform resource management over the open file.

```
#include <cstdio>
#include <memory>

using FileGuard = std::unique_ptr<FILE, int(*)(FILE*)>; ❶

void say_hello(FileGuard file❷) {
  fprintf(file.get(), "HELLO DAVE"); ❸
}

int main() {
  auto file = fopen("HAL9000", "w"); ❹
  if (!file) return errno; ❺
  FileGuard file_guard{ file, fclose }; ❻
  // File open here
  say_hello(std::move(file_guard)); ❼
  // File closed here
  return 0;
}
```

Listing 11-13: A program using a `std::unique_ptr` and a custom deleter to manage a file handle

This listing makes the `FileGuard` type alias ❶ for brevity. (Notice the deleter type matches the type of `fclose`.) Next is a `say_hello` function that takes a `FileGuard` by value ❷. Within `say_hello`, you `fprintf` `HELLO DAVE` to the `file` ❸. Because the lifetime of `file` is bound to `say_hello`, the file gets closed once `say_hello` returns. Within `main`, you open the file `HAL9000` in `w` mode, which will create or overwrite the file, and you save the raw `FILE*` file handle into `file` ❹. You check whether `file` is `nullptr`, indicating an error occurred, and return with `errno` if `HAL9000` couldn't be opened ❺. Next, you construct a `FileGuard` by passing the file handle `file` and the custom deleter `fclose` ❻. At this point, the file is open, and thanks to its custom deleter, `file_guard` manages the file's lifetime automatically.

To call `say_hello`, you need to transfer ownership into that function (because it takes a `FileGuard` by value) ❼. Recall from "Value Categories" on page 124 that variables like `file_guard` are lvalues. This means you must move it into `say_hello` with `std::move`, which writes `HELLO DAVE` to the file. If you omit `std::move`, the compiler would attempt to copy it into `say_hello`. Because `unique_ptr` has a deleted copy constructor, this would generate a compiler error.

When `say_hello` returns, its `FileGuard` argument destructs and the custom deleter calls `fclose` on the file handle. Basically, it's impossible to leak the file handle. You've tied it to the lifetime of `FileGuard`.

A Partial List of Supported Operations

Table 11-3 enumerates all the supported `std::unique_ptr` operations. In this table, `ptr` is a raw pointer, `u_ptr` is a unique pointer, and `del` is a deleter.

Table 11-3: All of the Supported `std::unique_ptr` Operations

Operation	Notes
`unique_ptr<...>{ }` or `unique_ptr<...>{ nullptr }`	Creates an empty unique pointer with a `std::default_delete<...>` deleter.
`unique_ptr<...>{ ptr }`	Creates a unique pointer owning the dynamic object pointed to by `ptr`. Uses a `std::default_delete<...>` deleter.
`unique_ptr<...>{ ptr, del }`	Creates a unique pointer owning the dynamic object pointed to by `ptr`. Uses `del` as deleter.
`unique_ptr<...>{ move(u_ptr) }`	Creates a unique pointer owning the dynamic object pointed to by the unique pointer `u_ptr`. Transfers ownership from `u_ptr` to the newly created unique pointer. Also moves the deleter of `u_ptr`.
`~unique_ptr<...>()`	Calls deleter on the owned object if full.
`u_ptr1 = move(u_ptr2)`	Transfers ownership of owned object and deleter from `u_ptr2` to `u_ptr1`. Destroys currently owned object if full.
`u_ptr1.swap(u_ptr2)`	Exchanges owned objects and deleters between `u_ptr1` and `u_ptr2`.
`swap(u_ptr1, u_ptr2)`	A free function identical to the `swap` method.
`u_ptr.reset()`	If full, calls deleter on object owned by `u_ptr`.
`u_ptr.reset(ptr)`	Deletes currently owned object; then takes ownership of `ptr`.
`ptr = u_ptr.release()`	Returns the raw pointer `ptr`; `u_ptr` becomes empty. Deleter is not called.
`ptr = u_ptr.get()`	Returns the raw pointer `ptr`; `u_ptr` retains ownership.
`*u_ptr`	Dereference operator on owned object.
`u_ptr->`	Member dereference operator on owned object.
`u_ptr[index]`	References the element at `index` (arrays only).
`bool{ u_ptr }`	`bool` conversion: `true` if full, `false` if empty.

`u_ptr1 == u_ptr2`
`u_ptr1 != u_ptr2`
`u_ptr1 > u_ptr2`
`u_ptr1 >= u_ptr2`
`u_ptr1 < u_ptr2`
`u_ptr1 <= u_ptr2`
	Comparison operators; equivalent to evaluating comparison operators on raw pointers.

`u_ptr.get_deleter()`	Returns a reference to the deleter.

Shared Pointers

A *shared pointer* has transferable, non-exclusive ownership over a single dynamic object. You can move shared pointers, which makes them transferable, and you *can* copy them, which makes their ownership non-exclusive.

Non-exclusive ownership means that a `shared_ptr` checks whether any other `shared_ptr` objects also own the object before destroying it. This way, the last owner is the one to release the owned object.

The stdlib has a `std::shared_ptr` available in the `<memory>` header, and Boost has a `boost::shared_ptr` available in the `<boost/smart_ptr/shared_ptr.hpp>` header. You'll use the stdlib version here.

NOTE *Both the stdlib and Boost* `shared_ptr` *are essentially identical, with the notable exception that Boost's shared pointer doesn't support arrays and requires you to use the* `boost::shared_array` *class in* `<boost/smart_ptr/shared_array.hpp>`*. Boost offers a shared pointer for legacy reasons, but you should use the stdlib shared pointer.*

Constructing

The `std::shared_ptr` pointer supports all the same constructors as `std::unique_ptr`. The default constructor yields an empty shared pointer. To instead establish ownership over a dynamic object, you can pass a pointer to the `shared_ptr` constructor, like so:

```
std::shared_ptr<int> my_ptr{ new int{ 808 } };
```

You also have a corollary `std::make_shared` template function that forwards arguments to the pointed-to type's constructor:

```
auto my_ptr = std::make_shared<int>(808);
```

You should generally use `make_shared`. Shared pointers require a *control block*, which keeps track of several quantities, including the number of shared owners. When you use `make_shared`, you can allocate the control block and the owned dynamic object simultaneously. If you first use `operator new` and then allocate a shared pointer, you're making two allocations instead of one.

NOTE *Sometimes you might want to avoid using* `make_shared`*. For example, if you'll be using a* `weak_ptr`*, you'll still need the control block even if you can deallocate the object. In such a situation, you might prefer to have two allocations.*

Because a control block is a dynamic object, `shared_ptr` objects sometimes need to allocate dynamic objects. If you wanted to take control over how `shared_ptr` allocates, you could override `operator new`. But this is shooting a sparrow with a cannon. A more tailored approach is to provide an optional template parameter called an *allocator type*.

Specifying an Allocator

The allocator is responsible for allocating, creating, destroying, and deallocating objects. The default allocator, `std::allocator`, is a template class defined in the `<memory>` header. The default allocator allocates memory from dynamic storage and takes a template parameter. (You'll learn about customizing this behavior with a user-defined allocator in "Allocators" on page 365).

Both the shared_ptr constructor and make_shared have an allocator type template parameter, making three total template parameters: the pointed-to type, the deleter type, and the allocator type. For complicated reasons, you only ever need to declare the *pointed-to type* parameter. You can think of the other parameter types as being deduced from the pointed-to type.

For example, here's a fully adorned make_shared invocation including a constructor argument, a custom deleter, and an explicit std::allocator:

```
std::shared_ptr<int❶> sh_ptr{
  new int{ 10 }❷,
  [](int* x) { delete x; } ❸,
  std::allocator<int>{} ❹
};
```

Here, you specify a single template parameter, int, for the pointed-to type ❶. In the first argument, you allocate and initialize an int ❷. Next is a custom deleter ❸, and as a third argument you pass a std::allocator ❹.

For technical reasons, you can't use a custom deleter or custom allocator with make_shared. If you want a custom allocator, you can use the sister function of make_shared, which is std::allocate_shared. The std::allocate_shared function takes an allocator as the first argument and forwards the remainder of the arguments to the owned object's constructor:

```
auto sh_ptr = std::allocate_shared<int❶>(std::allocator<int>{}❷, 10❸);
```

As with make_shared, you specify the owned type as a template parameter ❶, but you pass an allocator as the first argument ❷. The rest of the arguments forward to the constructor of int ❸.

NOTE *For the curious, here are two reasons why you can't use a custom deleter with* make_shared*. First,* make_shared *uses* new *to allocate space for the owned object and the control block. The appropriate deleter for* new *is* delete*, so generally a custom deleter wouldn't be appropriate. Second, the custom deleter can't generally know how to deal with the control block, only with the owned object.*

It isn't possible to specify a custom deleter with either make_shared or allocate_shared. If you want to use a custom deleter with shared pointers, you must use one of the appropriate shared_ptr constructors directly.

Supported Operations

The std::shared_ptr supports every operation that std::unique_ptr and boost::scoped_ptr support. You could use the following type alias as a drop-in replacement for ScopedOathbreakers in Listings 11-1 to 11-7 and UniqueOathbreakers from Listings 11-10 to 11-13:

```
using SharedOathbreakers = std::shared_ptr<DeadMenOfDunharrow>;
```

The major functional difference between a shared pointer and a unique pointer is that you can copy shared pointers.

Transferable, Non-Exclusive Ownership

Shared pointers are transferable (you *can* move them), and they have non-exclusive ownership (you *can* copy them). Listing 11-10, which illustrates a unique pointer's move semantics, works the same for a shared pointer. Listing 11-14 demonstrates that shared pointers also support copy semantics.

```
TEST_CASE("SharedPtr can be used in copy") {
  auto aragorn = std::make_shared<DeadMenOfDunharrow>();
  SECTION("construction") {
    auto son_of_arathorn{ aragorn }; ❶
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❷
  }
  SECTION("assignment") {
    SharedOathbreakers son_of_arathorn; ❸
    son_of_arathorn = aragorn; ❹
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❺
  }
  SECTION("assignment, and original gets discarded") {
    auto son_of_arathorn = std::make_shared<DeadMenOfDunharrow>(); ❻
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 2);  ❼
    son_of_arathorn = aragorn; ❽
    REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❾
  }
}
```

Listing 11-14: The `std::shared_ptr` supports copy.

After constructing the shared pointer `aragorn`, you have three tests. The first test illustrates that the copy constructor that you use to build `son_of_arathorn` ❶ shares ownership over the same `DeadMenOfDunharrow` ❷.

In the second test, you construct an empty shared pointer `son_of_arathorn` ❸ and then show that copy assignment ❹ also doesn't change the number of `DeadMenOfDunharrow` ❺.

The third test illustrates that when you construct the full shared pointer `son_of_arathorn` ❻, the number of `DeadMenOfDunharrow` increases to 2 ❼. When you copy assign `aragorn` to `son_of_arathorn` ❽, the `son_of_arathorn` deletes its `DeadMenOfDunharrow` because it has sole ownership. It then increments the reference count of the `DeadMenOfDunharrow` owned by `aragorn`. Because both shared pointers own the same `DeadMenOfDunharrow`, the `oaths_to_fulfill` decrements from 2 to 1 ❾.

Shared Arrays

A `shared array` is a shared pointer that owns a dynamic array and supports `operator[]`. It works just like a unique array except it has non-exclusive ownership.

Deleters

Deleters work the same way for shared pointers as they do for unique pointers except you don't need to provide a template parameter with the deleter's type. Simply pass the deleter as the second constructor argument. For example, to convert Listing 11-12 to use a shared pointer, you simply drop in the following type alias:

```
using FileGuard = std::shared_ptr<FILE>;
```

Now, you're managing `FILE*` file handles with shared ownership.

A Partial List of Supported Operations

Table 11-4 provides a mostly complete listing of the supported constructors of `shared_ptr`. In this table, `ptr` is a raw pointer, `sh_ptr` is a shared pointer, `u_ptr` is a unique pointer, `del` is a deleter, and `alc` is an allocator.

Table 11-4: All of the Supported `std::shared_ptr` Constructors

Operation	Notes
`shared_ptr<...>{ }` or `shared_ptr<...>{ nullptr }`	Creates an empty shared pointer with a `std::default_delete<T>` and a `std::allocator<T>`.
`shared_ptr<...>{ **ptr, [del], [alc]** }`	Creates a shared pointer owning the dynamic object pointed to by **`ptr`**. Uses a `std::default_delete<T>` and a `std::allocator<T>` by default; otherwise, **`del`** as deleter, **`alc`** as allocator if supplied.
`shared_ptr<...>{ **sh_ptr** }`	Creates a shared pointer owning the dynamic object pointed to by the shared pointer **`sh_ptr`**. Copies ownership from **`sh_ptr`** to the newly created shared pointer. Also copies the deleter and allocator of **`sh_ptr`**.
`shared_ptr<...>{ **sh_ptr , ptr** }`	An aliasing constructor: the resulting shared pointer holds an unmanaged reference to **`ptr`** but participates in **`sh_ptr`** reference counting.
`shared_ptr<...>{ move(**sh_ptr**) }`	Creates a shared pointer owning the dynamic object pointed to by the shared pointer **`sh_ptr`**. Transfers ownership from **`sh_ptr`** to the newly created shared pointer. Also moves the deleter of **`sh_ptr`**.
`shared_ptr<...>{ move(**u_ptr**) }`	Creates a shared pointer owning the dynamic object pointed to by the unique pointer **`u_ptr`**. Transfers ownership from **`u_ptr`** to the newly created shared pointer. Also moves the deleter of **`u_ptr`**.

Table 11-5 provides a listing of most of the supported operations of `std::shared_ptr`. In this table, `ptr` is a raw pointer, `sh_ptr` is a shared pointer, `u_ptr` is a unique pointer, `del` is a deleter, and `alc` is an allocator.

Table 11-5: Most of the Supported `std::shared_ptr` Operations

Operation	Notes
`~shared_ptr<...>()`	Calls deleter on the owned object if no other owners exist.
`sh_ptr1 = sh_ptr2`	Copies ownership of owned object and deleter from `sh_ptr2` to `sh_ptr1`. Increments number of owners by 1. Destroys currently owned object if no other owners exist.
`sh_ptr = move(u_ptr)`	Transfers ownership of owned object and deleter from `u_ptr` to `sh_ptr`. Destroys currently owned object if no other owners exist.
`sh_ptr1 = move(sh_ptr2)`	Transfers ownership of owned object and deleter from `sh_ptr2` to `sh_ptr1`. Destroys currently owned object if no other owners exist.
`sh_ptr1.swap(sh_ptr2)`	Exchanges owned objects and deleters between `sh_ptr1` and `sh_ptr2`.
`swap(sh_ptr1, sh_ptr2)`	A free function identical to the `swap` method.
`sh_ptr.reset()`	If full, calls deleter on object owned by `sh_ptr` if no other owners exist.
`sh_ptr.reset(ptr, [del], [alc])`	Deletes currently owned object if no other owners exist; then takes ownership of `ptr`. Can optionally provide deleter `del` and allocator `alc`. These default to `std::default_delete<T>` and `std::allocator<T>`.
`ptr = sh_ptr.get()`	Returns the raw pointer `ptr`; `sh_ptr` retains ownership.
`*sh_ptr`	Dereference operator on owned object.
`sh_ptr->`	Member dereference operator on owned object.
`sh_ptr.use_count()`	References the total number of shared pointers owning the owned object; zero if empty.
`sh_ptr[index]`	Returns the element at `index` (arrays only).
`bool{ sh_ptr }`	`bool` conversion: `true` if full, `false` if empty.

`sh_ptr1 == sh_ptr2`
`sh_ptr1 != sh_ptr2`
`sh_ptr1 > sh_ptr2`
`sh_ptr1 >= sh_ptr2`
`sh_ptr1 < sh_ptr2`
`sh_ptr1 <= sh_ptr2`
	Comparison operators; equivalent to evaluating comparison operators on raw pointers.

`sh_ptr.get_deleter()`	Returns a reference to the deleter.

Weak Pointers

A *weak pointer* is a special kind of smart pointer that has no ownership over the object to which it refers. Weak pointers allow you to track an object and to convert the weak pointer into a shared pointer *only if the tracked object still* *exists*. This allows you to generate temporary ownership over an object. Like shared pointers, weak pointers are movable and copyable.

A common usage for weak pointers is *caches*. In software engineering, a cache is a data structure that stores data temporarily so it can be retrieved faster. A cache could keep weak pointers to objects so they destruct once all other owners release them. Periodically, the cache can scan its stored weak pointers and trim those with no other owners.

The stdlib has a `std::weak_ptr`, and Boost has a `boost::weak_ptr`. These are essentially identical and are only meant to be used with their respective shared pointers, `std::shared_ptr` and `boost::shared_ptr`.

Constructing

Weak pointer constructors are completely different from scoped, unique, and shared pointers because weak pointers don't directly own dynamic objects. The default constructor constructs an empty weak pointer. To construct a weak pointer that tracks a dynamic object, you must construct it using either a shared pointer or another weak pointer.

For example, the following passes a shared pointer into the weak pointer's constructor:

```
auto sp = std::make_shared<int>(808);
std::weak_ptr<int> wp{ sp };
```

Now the weak pointer `wp` will track the object owned by the shared pointer `sp`.

Obtaining Temporary Ownership

Weak pointers invoke their `lock` method to get temporary ownership of their tracked object. The lock method always creates a shared pointer. If the tracked object is alive, the returned shared pointer owns the tracked object. If the tracked object is no longer alive, the returned shared pointer is empty. Consider the example in Listing 11-15.

```
TEST_CASE("WeakPtr lock() yields") {
  auto message = "The way is shut.";
  SECTION("a shared pointer when tracked object is alive") {
    auto aragorn = std::make_shared<DeadMenOfDunharrow>(message); ❶
    std::weak_ptr<DeadMenOfDunharrow> legolas{ aragorn }; ❷
    auto sh_ptr = legolas.lock(); ❸
    REQUIRE(sh_ptr->message == message); ❹
    REQUIRE(sh_ptr.use_count() == 2); ❺
  }
  SECTION("empty when shared pointer empty") {
    std::weak_ptr<DeadMenOfDunharrow> legolas;
    {
      auto aragorn = std::make_shared<DeadMenOfDunharrow>(message); ❻
      legolas = aragorn; ❼
    }
```

```
    auto sh_ptr = legolas.lock(); ❽
    REQUIRE(nullptr == sh_ptr); ❾
  }
}
```

Listing 11-15: The `std::weak_ptr` exposes a `lock` method for obtaining temporary ownership.

In the first test, you create the shared pointer `aragorn` ❶ with a message. Next, you construct a weak pointer `legolas` using `aragorn` ❷. This sets up `legolas` to track the dynamic object owned by `aragorn`. When you call `lock` on the weak pointer ❸, `aragorn` is still alive, so you obtain the shared pointer `sh_ptr`, which also owns the same `DeadMenOfDunharrow`. You confirm this by asserting that the `message` is the same ❹ and that the *use count* is 2 ❺.

In the second test, you also create an `aragorn` shared pointer ❻, but this time you use the assignment operator ❼, so the previously empty weak pointer `legolas` now tracks the dynamic object owned by `aragorn`. Next, `aragorn` falls out of block scope and dies. This leaves `legolas` tracking a dead object. When you call `lock` at this point ❽, you obtain an empty shared pointer ❾.

Advanced Patterns

In some advanced usages of shared pointers, you might want to create a class that allows instances to create shared pointers referring to themselves. The `std::enable_shared_from_this` class template implements this behavior. All that's required from a user perspective is to inherit from `enable_shared_from_this` in the class definition. This exposes the `shared_from_this` and `weak_from_this` methods, which produce either a `shared_ptr` or a `weak_ptr` referring to the current object. This is a niche case, but if you want to see more details, refer to [util.smartptr.enab].

Supported Operations

Table 11-6 lists most of the supported weak pointer operations. In this table, `w_ptr` is a weak pointer, and `sh_ptr` is a shared pointer.

Table 11-6: Most of the Supported `std::shared_ptr` Operations

Operation	Notes
`weak_ptr<...>{ }`	Creates an empty weak pointer.
`weak_ptr<...>{ w_ptr }` or `weak_ptr<...>{ sh_ptr }`	Tracks the object referred to by weak pointer **`w_ptr`** or shared pointer **`sh_ptr`**.
`weak_ptr<...>{ move(w_ptr) }`	Tracks the object referred to by **`w_ptr`**; then empties **`w_ptr`**.
`~weak_ptr<...>()`	Has no effect on the tracked object.
`w_ptr1` `=` **`sh_ptr`** or **`w_ptr1`** `=` **`w_ptr2`**	Replaces currently tracked object with the object owned by **`sh_ptr`** or tracked by **`w_ptr2`**.
`w_ptr1` `= move(`**`w_ptr2`**`)`	Replaces currently tracked object with object tracked by **`w_ptr2`**. Empties **`w_ptr2`**.

Operation	Notes
sh_ptr = **w_ptr**.lock()	Creates the shared pointer **sh_ptr** owning the object tracked by **w_ptr**. If the tracked object has expired, **sh_ptr** is empty.
w_ptr1.swap(**w_ptr2**)	Exchanges tracked objects between **w_ptr1** and **w_ptr2**.
swap(**w_ptr1, w_ptr2**)	A free function identical to the swap method.
w_ptr.reset()	Empties the weak pointer.
w_ptr.use_count()	Returns the number of shared pointers owning the tracked object.
w_ptr.expired()	Returns true if the tracked object has expired, false if it hasn't.
sh_ptr.use_count()	Returns the total number of shared pointers owning the owned object; zero if empty.

Intrusive Pointers

An *intrusive pointer* is a shared pointer to an object with an embedded reference count. Because shared pointers usually keep reference counts, they're not suitable for owning such objects. Boost provides an implementation called boost::intrusive_ptr in the <boost/smart_ptr/intrusive_ptr.hpp> header.

It's rare that a situation calls for an intrusive pointer. But sometimes you'll use an operating system or a framework that contains embedded references. For example, in Windows COM programming an intrusive pointer can be very useful: COM objects that inherit from the IUnknown interface have an AddRef and a Release method, which increment and decrement an embedded reference count (respectively).

Each time an intrusive_ptr is created, it calls the function intrusive_ptr_add_ref. When an intrusive_ptr is destroyed, it calls the intrusive_ptr_release free function. You're responsible for freeing appropriate resources in intrusive_ptr_release when the reference count falls to zero. To use intrusive_ptr, you must provide a suitable implementation of these functions.

Listing 11-16 demonstrates intrusive pointers using the DeadMenOfDunharrow class. Consider the implementations of intrusive_ptr_add_ref and intrusive_ptr_release in this listing.

```
#include <boost/smart_ptr/intrusive_ptr.hpp>

using IntrusivePtr = boost::intrusive_ptr<DeadMenOfDunharrow>; ❶
size_t ref_count{}; ❷

void intrusive_ptr_add_ref(DeadMenOfDunharrow* d) {
  ref_count++; ❸
}

void intrusive_ptr_release(DeadMenOfDunharrow* d) {
```

```
  ref_count--; ❹
  if (ref_count == 0) delete d; ❺
}
```

Listing 11-16: Implementations of `intrusive_ptr_add_ref` and `intrusive_ptr_release`

Using the type alias `IntrusivePtr` saves some typing ❶. Next, you declare a `ref_count` with static storage duration ❷. This variable keeps track of the number of living intrusive pointers. In `intrusive_ptr_add_ref`, you increment `ref_count` ❸. In `intrusive_ptr_release`, you decrement `ref_count` ❹. When `ref_count` drops to zero, you delete the `DeadMenOfDunharrow` argument ❺.

NOTE *It's absolutely critical that you use only a single `DeadMenOfDunharrow` dynamic object with intrusive pointers when using the setup in Listing 11-16. The `ref_count` approach will correctly track only a single object. If you have multiple dynamic objects owned by different intrusive pointers, the `ref_count` will become invalid, and you'll get incorrect `delete` behavior ❺.*

Listing 11-17 shows how to use the setup in Listing 11-16 with intrusive pointers.

```
TEST_CASE("IntrusivePtr uses an embedded reference counter.") {
  REQUIRE(ref_count == 0); ❶
  IntrusivePtr aragorn{ new DeadMenOfDunharrow{} }; ❷
  REQUIRE(ref_count == 1); ❸
  {
    IntrusivePtr legolas{ aragorn }; ❹
    REQUIRE(ref_count == 2); ❺
  }
  REQUIRE(DeadMenOfDunharrow::oaths_to_fulfill == 1); ❻
}
```

Listing 11-17: Using a `boost::intrusive_ptr`

This test begins by checking that `ref_count` is zero ❶. Next, you construct an intrusive pointer by passing a dynamically allocated `DeadMenOfDunharrow` ❷. This increases `ref_count` to 1, because creating an intrusive pointer invokes `intrusive_ptr_add_ref` ❸. Within a block scope, you construct another intrusive pointer `legolas` that shares ownership with `aragorn` ❹. This increases the `ref_count` to 2 ❺, because creating an intrusive pointer invokes `intrusive_ptr_add_ref`. When `legolas` falls out of block scope, it destructs, causing `intrusive_ptr_release` to invoke. This decrements `ref_count` to 1 but doesn't cause the owned object to delete ❻.

Summary of Smart Pointer Options

Table 11-7 summarizes all the smart pointer options available to use in stdlib and Boost.

Table 11-7: Smart Pointers in stdlib and Boost

Type name	stdlib header	Boost header	Movable/ transferable ownership	Copyable/ non-exclusive ownership
scoped_ptr		<boost/smart_ptr/scoped_ptr.hpp>		
scoped_array		<boost/smart_ptr/scoped_array.hpp>		
unique_ptr	<memory>		✓	
shared_ptr	<memory>	<boost/smart_ptr/shared_ptr.hpp>	✓	✓
shared_array		<boost/smart_ptr/shared_array.hpp>	✓	✓
weak_ptr	<memory>	<boost/smart_ptr/weak_ptr.hpp>	✓	✓
intrusive_ptr		<boost/smart_ptr/intrusive_ptr.hpp>	✓	✓

Allocators

Allocators are low-level objects that service requests for memory. The stdlib and Boost libraries enable you to provide allocators to customize how a library allocates dynamic memory.

In the majority of cases, the default allocator `std::allocate` is totally sufficient. It allocates memory using `operator new(size_t)`, which allocates raw memory from the free store, also known as the heap. It deallocates memory using `operator delete(void*)`, which deallocates the raw memory from the free store. (Recall from "Overloading Operator `new`" on page 189 that `operator new` and `operator delete` are defined in the `<new>` header.)

In some settings, such as gaming, high-frequency trading, scientific analyses, and embedded applications, the memory and computational overhead associated with the default free store operations is unacceptable. In such settings, it's relatively easy to implement your own allocator. Note that you really shouldn't implement a custom allocator unless you've conducted some performance testing that indicates that the default allocator is a bottleneck. The idea behind a custom allocator is that you know a lot more about your specific program than the designers of the default allocator model, so you can make improvements that will increase allocation performance.

At a minimum, you need to provide a template class with the following characteristics for it to work as an allocator:

- An appropriate default constructor
- A `value_type` member corresponding to the template parameter
- A template constructor that can copy an allocator's internal state while dealing with a change in `value_type`
- An `allocate` method
- A `deallocate` method
- An `operator==` and an `operator!=`

The `MyAllocator` class in Listing 11-18 implements a simple, pedagogical variant of `std::allocate` that keeps track of how many allocations and deallocations you've made.

```
#include <new>

static size_t n_allocated, n_deallocated;

template <typename T>
struct MyAllocator {
  using value_type = T; ❶
  MyAllocator() noexcept{ } ❷
  template <typename U>
  MyAllocator(const MyAllocator<U>&) noexcept { } ❸
  T* allocate(size_t n) { ❹
    auto p = operator new(sizeof(T) * n);
    ++n_allocated;
    return static_cast<T*>(p);
  }
  void deallocate(T* p, size_t n) { ❺
    operator delete(p);
    ++n_deallocated;
  }
};

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

Listing 11-18: A `MyAllocator` class modeled after `std::allocate`

First, you declare the `value_type` type alias for `T`, one of the requirements for implementing an allocator ❶. Next is a default constructor ❷ and a template constructor ❸. Both of these are empty because the allocator doesn't have state to pass on.

The `allocate` method ❹ models `std::allocate` by allocating the requisite number of bytes, `sizeof(T) * n`, using `operator new`. Next, it increments the static variable `n_allocated` so you can keep track of the number of allocations for testing purposes. The `allocate` method then returns a pointer to the newly allocated memory after casting `void*` to the relevant pointer type.

The `deallocate` method ❺ also models `std::allocate` by calling `operator delete`. As an analogy to `allocate`, it increments the `n_deallocated` static variable for testing and returns.

The final task is to implement an `operator==` and an `operator!=` taking the new class template. Because the allocator has no state, any instance is the same as any other instance, so `operator==` returns `true` ❻ and `operator!=` returns `true` ❼.

NOTE *Listing 11-18 is a teaching tool and doesn't actually make allocations any more efficient. It simply wraps the call to* `new` *and* `delete`.

So far, the only class you know about that uses an allocator is `std::shared_ptr`. Consider how Listing 11-19 uses `MyAllocator` with `std::allocate shared`.

```
TEST_CASE("Allocator") {
  auto message = "The way is shut.";
  MyAllocator<DeadMenOfDunharrow> alloc; ❶
  {
    auto aragorn = std::allocate_shared<DeadMenOfDunharrow>(my_alloc❷,
                                                             message❸);
    REQUIRE(aragorn->message == message); ❹
    REQUIRE(n_allocated == 1); ❺
    REQUIRE(n_deallocated == 0); ❻
  }
  REQUIRE(n_allocated == 1); ❼
  REQUIRE(n_deallocated == 1); ❽
}
```

Listing 11-19: Using `MyAllocator` *with* `std::shared_ptr`

You create a `MyAllocator` instance called `alloc` ❶. Within a block, you pass `alloc` as the first argument to `allocate_shared` ❷, which creates the shared pointer `aragorn` containing a custom `message` ❸. Next, you confirm that `aragorn` contains the correct `message` ❹, `n_allocated` is 1 ❺, and `n_deallocated` is 0 ❻.

After `aragorn` falls out of block scope and destructs, you verify that `n_allocated` is still 1 ❼ and `n_deallocated` is now 1 ❽.

NOTE *Because allocators handle low-level details, you can really get down into the weeds when specifying their behavior. See [allocator.requirements] in the ISO C++ 17 Standard for a thorough treatment.*

Summary

Smart pointers manage dynamic objects via RAII, and you can provide allocators to customize dynamic memory allocation. Depending on which smart pointer you choose, you can encode different ownership patterns onto the dynamic object.

EXERCISES

11-1. Reimplement Listing 11-12 to use a std::shared_ptr rather than a std::unique_ptr. Notice that although you've relaxed the ownership requirements from exclusive to non-exclusive, you're still transferring ownership to the say_hello function.

11-2. Remove the std::move from the call to say_hello. Then make an additional call to say_hello. Notice that the ownership of file_guard is no longer *transferred* to say_hello. This permits multiple calls.

11-3. Implement a Hal class that accepts a std::shared_ptr<FILE> in its constructor. In Hal's destructor, write the phrase Stop, Dave. to the file handle held by your shared pointer. Implement a write_status function that writes the phrase I'm completely operational. to the file handle. Here's a class declaration you can work from:

```
struct Hal {
  Hal(std::shared_ptr<FILE> file);
  ~Hal();
  void write_status();
  std::shared_ptr<FILE> file;
};
```

11-4. Create several Hal instances and invoke write_status on them. Notice that you don't need to keep track of how many Hal instances are open: file management gets handled via the shared pointer's shared ownership model.

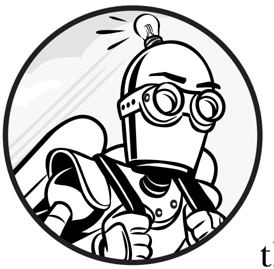
FURTHER READING

- *ISO International Standard ISO/IEC (2017) — Programming Language C++* (International Organization for Standardization; Geneva, Switzerland; *https://isocpp.org/std/the-standard/*)
- *The C++ Programming Language*, 4th Edition, by Bjarne Stroustrup (Pearson Education, 2013)
- *The Boost C++ Libraries*, 2nd Edition, by Boris Schäling (XML Press, 2014)
- *The C++ Standard Library: A Tutorial and Reference*, 2nd Edition, by Nicolai M. Josuttis (Addison-Wesley Professional, 2012)

12

UTILITIES

"See, the world is full of things more powerful than us. But if you know how to catch a ride, you can go places," Raven says. "Right. I'm totally hip to what you're saying."
—Neal Stephenson, Snow Crash



The stdlib and Boost libraries provide a throng of types, classes, and functions that satisfy common programming needs. Together, this motley collection of tools is called *utilities*. Aside from their small, uncomplicated, and focused nature, utilities vary functionally.

In this chapter, you'll learn about several simple data structures that handle many routine situations where you need objects to contain other objects. A discussion of dates and times follows, including coverage of several provisions for encoding calendars and clocks and for measuring elapsed time. The chapter wraps up with a trek through many numerical and mathematical tools available to you.

NOTE *The discussions of dates/times and numerics/math will be of great interest to certain readers and of only passing interest to others. If you are in the latter category, feel free to skim these sections.*

Data Structures

Between them, the stdlib and Boost libraries provide a venerable collection of useful data structures. A *data structure* is a type that stores objects and permits some set of operations over those stored objects. There is no magic compiler pixie dust that makes the utility data structures in this section work; you could implement your own versions with sufficient time and effort. But why reinvent the wheel?

tribool

The *tribool* is a `bool`-like type that supports three states rather than two: true, false, and indeterminate. Boost offers `boost::logic::tribool` in the `<boost/logic/tribool.hpp>` header. Listing 12-1 demonstrates how to initialize Boost a `tribool` using `true`, `false`, and the `boost::logic::indeterminate` type.

```
#include <boost/logic/tribool.hpp>

using boost::logic::indeterminate; ❶
boost::logic::tribool t = true❷, f = false❸, i = indeterminate❹;
```

Listing 12-1: Initializing Boost `tribool`

For convenience, a `using` declaration pulls in `indeterminate` from `boost::logic` ❶. Then you initialize the `tribool` `t` equal to `true` ❷, `f` equal to `false` ❸, and `i` equal to `indeterminate` ❹.

The `tribool` class implicitly converts to `bool`. If a `tribool` is `true`, it converts to `true`; otherwise, it converts to `false`. The `tribool` class also supports `operator!`, which returns `true` if `tribool` is `false`; otherwise, it returns `false`. Finally, `indeterminate` supports `operator()`, which takes a single `tribool` argument and returns `true` if that argument is `indeterminate`; otherwise, it returns `false`.

Listing 12-2 samples these Boolean conversions.

```
TEST_CASE("Boost tribool converts to bool") {
  REQUIRE(t); ❶
  REQUIRE_FALSE(f); ❷
  REQUIRE(!f); ❸
  REQUIRE_FALSE(!t); ❹
  REQUIRE(indeterminate(i)); ❺
  REQUIRE_FALSE(indeterminate(t)); ❻
}
```

Listing 12-2: Converting a `tribool` to a `bool`

This test demonstrates the basic results from `bool` conversion ❶❷, `operator!` ❸❹, and `indeterminate` ❺❻.

Boolean Operations

The `tribool` class supports all the Boolean operators. Whenever a `tribool` expression doesn't involve an `indeterminate` value, the result is the same as the equivalent Boolean expression. Whenever an `indeterminate` is involved, the result can be `indeterminate`, as Listing 12-3 illustrates.

```
TEST_CASE("Boost Tribool supports Boolean operations") {
  auto t_or_f = t || f;
  REQUIRE(t_or_f); ❶
  REQUIRE(indeterminate(t && indeterminate)); ❷
  REQUIRE(indeterminate(f || indeterminate)); ❸
  REQUIRE(indeterminate(!i)); ❹
}
```

Listing 12-3: The `boost::tribool` supports Boolean operations.

Because neither `t` nor `f` is indeterminate, `t || f` evaluates just like an ordinary Boolean expression, so `t_or_f` is `true` ❶. Boolean expressions that involve an indeterminate can be indeterminate. Boolean AND ❷, OR ❸, and NOT ❹ evaluate to `indeterminate` if there isn't enough information.

When to Use tribool

Aside from describing the vital status of Schrödinger's cat, you can use `tribool` in settings in which operations can take a long time. In such settings, a `tribool` could describe whether the operation was successful. An `indeterminate` value could model that the operation is still pending.

The `tribool` class makes for neat, concise `if` statements, as shown in Listing 12-4.

```
TEST_CASE("Boost Tribool works nicely with if statements") {
  if (i) FAIL("Indeterminate is true."); ❶
  else if (!i) FAIL("Indeterminate is false."); ❷
  else {} // OK, indeterminate ❸
}
```

Listing 12-4: Using an `if` statement with `tribool`

The first expression ❶ evaluates only if the `tribool` is `true`, the second expression ❷ evaluates only if it's `false`, and the third only executes in the `indeterminate` case ❸.

NOTE *The mere mention of a `tribool` might have caused you to scrunch up your face in disgust. Why, you might ask, couldn't you just use an integer where 0 is false, 1 is true, and any other value is indeterminate? You could, but consider that the `tribool` type supports all the usual Boolean operations while correctly propagating indeterminate values. Again, why reinvent the wheel?*

A Partial List of Supported Operations

Table 12-1 provides a list of the most supported `boost::tribool` operations. In this table, `tb` is a `boost::tribool`.

Table 12-1: The Most Supported `boost::tribool` Operations

Operation	Notes
`tribool{}` `tribool{ false }`	Constructs a `tribool` with value `false`.
`tribool{ true }`	Constructs a `tribool` with value `true`.
`tribool{ indeterminate }`	Constructs a `tribool` with value `indeterminate`.
tb`.safe_bool()`	Evaluates to `true` if **tb** is `true`, else `false`.
`indeterminate(`**tb**`)`	Evaluates to `true` if **tb** is `indeterminate`, else `false`.
`!`**tb**	Evaluates to `true` if **tb** is `false`, else `false`.
tb1 `&&` **tb2**	Evaluates to `true` if **tb1** and **tb2** are `true`; evaluates to `false` if **tb1** or **tb2** are `false`; otherwise, `indeterminate`.
tb1 || **tb2**	Evaluates to `true` if **tb1** or **tb2** are `true`; evaluates to `false` if **tb1** and **tb2** are `false`; otherwise, `indeterminate`.
`bool{` **tb** `}`	Evaluates to `true` if **tb** is true, else `false`.

optional

An *optional* is a class template that contains a value that might or might not be present. The primary use case for an `optional` is the return type of a function that might fail. Rather than throwing an exception or returning multiple values, a function can instead return an `optional` that will contain a value if the function succeeded.

The stdlib has `std::optional` in the <optional> header, and Boost has `boost::optional` in the <boost/optional.hpp> header.

Consider the setup in Listing 12-5. The function `take` wants to return an instance of `TheMatrix` only if you take a `Pill::Blue`; otherwise, `take` returns a `std::nullopt`, which is a stdlib-provided constant `std::optional` type with uninitialized state.

```
#include <optional>

struct TheMatrix { ❶
  TheMatrix(int x) : iteration { x } { }
  const int iteration;
};

enum Pill { Red, Blue }; ❷

std::optional<TheMatrix>❸ take(Pill pill❹) {
  if(pill == Pill::Blue) return TheMatrix{ 6 }; ❺
  return std::nullopt; ❻
}
```

Listing 12-5: A take function returning a `std::optional`

The `TheMatrix` type takes a single `int` constructor argument and stores it into the `iteration` member ❶. The `enum` called `Pill` takes the values `Red` and

Blue ❷. The take function returns a std::optional<TheMatrix> ❸ and accepts a single Pill argument ❹. If you pass Pill::Blue to the take function, it returns a TheMatrix instance ❺; otherwise, it returns a std::nullopt ❻.

First, consider Listing 12-6, where you take the blue pill.

```
TEST_CASE("std::optional contains types") {
  if (auto matrix_opt = take(Pill::Blue)) { ❶
    REQUIRE(matrix_opt->iteration == 6); ❷
    auto& matrix = matrix_opt.value();
    REQUIRE(matrix.iteration == 6); ❸
  } else {
    FAIL("The optional evaluated to false.");
  }
}
```

Listing 12-6: A test exploring the std::optional *type with* Pill::Blue

You take the blue pill, which results in the std::optional result containing an initialized TheMatrix, so the if statement's conditional expression evaluates to true ❶. Listing 12-6 also demonstrates the use of operator-> ❷ and value() ❸ to access the underlying value.

What happens when you take the red pill? Consider Listing 12-7.

```
TEST_CASE("std::optional can be empty") {
  auto matrix_opt = take(Pill::Red); ❶
  if (matrix_opt) FAIL("The Matrix is not empty."); ❷
  REQUIRE_FALSE(matrix_opt.has_value()); ❸
}
```

Listing 12-7: A test exploring the std::optional *type with* Pill::Red

You take the red pill ❶, and the resulting matrix_opt is empty. This means matrix_opt converts to false ❷ and has_value() also returns false ❸.

A Partial List of Supported Operations

Table 12-2 provides a list of the most supported std::optional operations. In this table, opt is a std::optional<T> and t is an object of type T.

Table 12-2: The Most Supported std::optional Operations

Operation	Notes
optional<T>{} optional<T>{std::nullopt}	Constructs an empty optional.
optional<T>{ **opt** }	Copy constructs an optional from **opt**.
optional<T>{ move(**opt**) }	Move constructs an optional from **opt**, which is empty after the constructor completes.
optional<T>{ **t** } **opt** = **t**	Copies **t** into optional.
optional<T>{ move(**t**) } **opt** = move(**t**)	Moves **t** into optional.

(continued)

Table 12-2: The Most Supported `std::optional` Operations (continued)

Operation	Notes
`opt->mbr`	Member dereference; accesses the `mbr` member of object contained by `opt`.
`*opt` `opt.value()`	Returns a reference to the object contained by `opt`; `value()` checks for empty and throws `bad_optional_access`.
`opt.value_or(T{ ... })`	If `opt` contains an object, returns a copy; else returns the argument.
`bool{ opt }` `opt.has_value()`	Returns `true` if `opt` contains an object, else `false`.
`opt1.swap(opt2)` `swap(opt1, opt2)`	Swaps the objects contained by `opt1` and `opt2`.
`opt.reset()`	Destroys object contained by `opt`, which is empty after `reset`.
`opt.emplace(...)`	Constructs a type in place, forwarding all arguments to the appropriate constructor.
`make_optional<T>(...)`	Convenience function for constructing an `optional`; forwards arguments to the appropriate constructor.

`opt1 == opt2`
`opt1 != opt2`
`opt1 > opt2`
`opt1 >= opt2`
`opt1 < opt2`
`opt1 <= opt2`
	When evaluating equality of two `optional` objects, `true` if both are empty or if both contain objects and those objects are equal; else `false`. For comparison, an empty `optional` is always less than an optional containing a value. Otherwise, the result is the comparison of the contained types.

pair

A *pair* is a class template that contains two objects of different types in a single object. The objects are ordered, and you can access them via the members `first` and `second`. A `pair` supports comparison operators, has defaulted copy/move constructors, and works with structured binding syntax.

The stdlib has `std::pair` in the `<utility>` header, and Boost has `boost::pair` in the `<boost/pair.hpp>` header.

NOTE *Boost also has `boost::compressed_pair` available in the `<boost/compressed_pair.hpp>` header. It's slightly more efficient when one of the members is empty.*

First, you create some simple types to make a pair out of, such as the simple `Socialite` and `Valet` classes in Listing 12-8.

```
#include <utility>

struct Socialite { const char* birthname; };
struct Valet { const char* surname; };
Socialite bertie{ "Wilberforce" };
Valet reginald{ "Jeeves" };
```

Listing 12-8: The `Socialite` and `Valet` classes

Now that you have a Socialite and a Valet, bertie and reginald, you can construct a std::pair and experiment with extracting elements. Listing 12-9 uses the first and second members to access the contained types.

```
TEST_CASE("std::pair permits access to members") {
  std::pair<Socialite, Valet> inimitable_duo{ bertie, reginald }; ❶
  REQUIRE(inimitable_duo.first.birthname == bertie.birthname); ❷
  REQUIRE(inimitable_duo.second.surname == reginald.surname); ❸
}
```

Listing 12-9: The `std::pair` supports member extraction.

You construct a std::pair by passing in the objects you want to copy ❶. You use the first and second members of std::pair to extract the Socialite ❷ and Valet ❸ out of inimitable_duo. Then you can compare the birthname and surname members of these to their originals.

Listing 12-10 shows std::pair member extraction and structured binding syntax.

```
TEST_CASE("std::pair works with structured binding") {
  std::pair<Socialite, Valet> inimitable_duo{ bertie, reginald };
  auto& [idle_rich, butler] = inimitable_duo; ❶
  REQUIRE(idle_rich.birthname == bertie.birthname); ❷
  REQUIRE(butler.surname == reginald.surname); ❸
}
```

Listing 12-10: The `std::pair` supports structured binding syntax.

Here you use the structured binding syntax ❶ to extract references to the first and second members of inimitable_duo into idle_rich and butler. As in Listing 12-9, you ensure that the birthname ❷ and surname ❸ match the originals.

A Partial List of Supported Operations

Table 12-3 provides a list of the most supported std::pair operations. In this table, pr is a std::pair<A, B>, a is an object of type A, and b is an object of type B.

Table 12-3: The Most Supported std::pair Operations

Operation	Notes
pair<...>{}	Constructs an empty pair.
pair<...>{ **pr** }	Copy constructs from **pr**.
pair<...>{ move(**pr**) }	Move constructs from **pr**.
pair<...>{ **a**, **b** }	Constructs a pair by copying **a** and **b**.
pair<...>{ move(**a**), move(**b**) }	Constructs a pair by moving **a** and **b**.
pr1 = **pr2**	Copy assigns from **pr2**.
pr1 = move(**pr2**)	Move assigns from **pr2**.

(continued)

Table 12-3: The Most Supported `std::pair` Operations (continued)

Operation	Notes
`pr.first` `get<0>(pr)`	Returns a reference to the `first` element.
`pr.second` `get<1>(pr)`	Returns a reference to the `second` element.
`get<T>(pr)`	If `first` and `second` have different types, returns a reference to the element of type `T`.
`pr1.swap(pr2)` `swap(pr1, pr2)`	Swaps the objects contained by `pr1` and `pr2`.
`make_pair<...>(a, b)`	Convenience function for constructing a `pair`.

`pr1 == pr2`
`pr1 != pr2`
`pr1 > pr2`
`pr1 >= pr2`
`pr1 < pr2`
`pr1 <= pr2`
	Equal if both `first` and `second` are equal.
	Greater than/less than comparisons begin with `first`. If `first` members are equal, compare `second` members.

tuple

A *tuple* is a class template that takes an arbitrary number of heterogeneous elements. It's a generalization of `pair`, but a `tuple` doesn't expose its members as `first`, `second`, and so on like a `pair`. Instead, you use the non-member function template `get` to extract elements.

The stdlib has `std::tuple` and `std::get` in the `<tuple>` header, and Boost has `boost::tuple` and `boost::get` in the `<boost/tuple/tuple.hpp>` header.

Let's add a third class, `Acquaintance`, to test a `tuple`:

```
struct Acquaintance { const char* nickname; };
Acquaintance hildebrand{ "Tuppy" };
```

To extract these elements, you have two modes of using `get`. In the primary case, you can always provide a template parameter corresponding to the zero-based index of the element you want to extract. In the event the `tuple` doesn't contain elements with the same types, you can alternatively provide a template parameter corresponding to the type of the element you want to extract, as Listing 12-11 illustrates.

```
TEST_CASE("std::tuple permits access to members with std::get") {
  using Trio = std::tuple<Socialite, Valet, Acquaintance>;
  Trio truculent_trio{ bertie, reginald, hildebrand };
  auto& bertie_ref = std::get<0>(truculent_trio); ❶
  REQUIRE(bertie_ref.birthname == bertie.birthname);

  auto& tuppy_ref = std::get<Acquaintance>(truculent_trio); ❷
  REQUIRE(tuppy_ref.nickname == hildebrand.nickname);
}
```

Listing 12-11: A `std::tuple` supports member extraction and structured binding syntax.

You can build a std::tuple in an analogous way to how you built a std::pair. First, you extract the Socialite member with get<0> ❶. Because Socialite is the first template parameter, you use 0 for the std::get template parameter. Then you extract the Acquaintance member with std::get<Acquaintance> ❷. Because there's only one element of type Acquaintance, you're permitted to use this mode of get access.

Like pair, tuple also allows structured binding syntax.

A Partial List of Supported Operations

Table 12-4 provides a list of the most supported std::tuple operations. In this table, tp is a std::tuple<A, B>, a is an object of type A, and b is an object of type B.

Table 12-4: The Most Supported std::tuple Operations

Operation	Notes
`tuple<...>{ [alc] }`	Constructs an empty tuple. Uses std::allocate as default allocator **alc**.
`tuple<...>{ [alc], tp }`	Copy constructs from **tp**. Uses std::allocate as default allocator **alc**.
`tuple<...>{ [alc],move(tp) }`	Move constructs from **tp**. Uses std::allocate as default allocator **alc**.
`tuple<...>{ [alc], a, b }`	Constructs a tuple by copying **a** and **b**. Uses std::allocate as default allocator **alc**.
`tuple<...>{ [alc], move(a), move(b) }`	Constructs a tuple by moving **a** and **b**. Uses std::allocate as default allocator **alc**.
`tp1 = tp2`	Copy assigns from **tp2**.
`tp1 = move(tp2)`	Move assigns from **tp2**.
`get<i>(tp)`	Returns a reference to the **i**th element (zero-based).
`get<T>(tp)`	Returns a reference to the element of type **T**. Fails to compile if more than one element share this type.
`tp1.swap(tp2)` `swap(tp1, tp2)`	Swaps the objects contained by **tp1** and **tp2**.
`make_tuple<...>(a, b)`	Convenience function for constructing a tuple.
`tuple_cat<...>(tp1, tp2)`	Concatenates all the tuples passed in as arguments.

`tp1 == tp2`
`tp1 != tp2`
`tp1 > tp2`
`tp1 >= tp2`
`tp1 < tp2`
`tp1 <= tp2`
	Equal if all elements are equal.
	Greater than/less than comparisons proceed from first element to last.

any

An *any* is a class that stores single values of any type. It is *not* a class template. To convert an any into a concrete type, you use an *any cast*, which is a non-member function template. Any cast conversions are type safe; if you attempt to cast an any and the type doesn't match, you get an exception. With any, you can perform some kinds of generic programming *without templates*.

The stdlib has `std::any` in the `<any>` header, and Boost has `boost::any` in the `<boost/any.hpp>` header.

To store a value into an any, you use the `emplace` method template. It takes a single template parameter corresponding to the type you want to store into any (the *storage type*). Any arguments you pass into `emplace` get forwarded to an appropriate constructor for the given storage type. To extract the value, you use `any_cast`, which takes a template parameter corresponding to the current storage type of any (called the *state* of any). You pass the any as the sole parameter to `any_cast`. As long as the state of any matches the template parameter, you get the desired type out. If the state doesn't match, you get a `bad_any_cast` exception.

Listing 12-12 illustrates these basic interactions with a `std::any`.

```
#include <any>

struct EscapeCapsule {
  EscapeCapsule(int x) : weight_kg{ x } { }
  int weight_kg;
}; ❶

TEST_CASE("std::any allows us to std::any_cast into a type") {
  std::any hagunemnon; ❷
  hagunemnon.emplace<EscapeCapsule>(600); ❸
  auto capsule = std::any_cast<EscapeCapsule>(hagunemnon); ❹
  REQUIRE(capsule.weight_kg == 600);
  REQUIRE_THROWS_AS(std::any_cast<float>(hagunemnon), std::bad_any_cast); ❺
}
```

Listing 12-12: The `std::any` and `std::any_cast` allow you to extract concrete types.

You declare the `EscapeCapsule` class ❶. Within the test, you construct an empty `std::any` called `hagunemnon` ❷. Next, you use `emplace` to store an `EscapeCapsule` with `weight_kg = 600` ❸. You can extract the `EscapeCapsule` back out using `std::any_cast` ❹, which you store into a new `EscapeCapsule` called `capsule`. Finally, you show that attempting to invoke `any_cast` to cast the `hagunemnon` into a `float` results in a `std::bad_any_cast` exception ❺.

A Partial List of Supported Operations

Table 12-5 provides a list of the most supported `std::any` operations. In this table, `ay` is a `std::any` and `t` is an object of type `T`.

Table 12-5: The Most Supported `std::any` Operations

Operation	Notes
`any{}`	Constructs an empty any object.
`any{ ay }`	Copy constructs from **ay**.
`any{ move(ay) }`	Move constructs from **ay**.
`any{ move(t) }`	Constructs an any object containing an in-place constructed object from **t**.
`ay = t`	Destructs the object currently contained by **ay**; copies **t**.
`ay = move(t)`	Destructs the object currently contained by **ay**; moves **t**.
`ay1 = ay2`	Copy assigns from **ay2**.
`ay1 = move(ay2)`	Move assigns from **ay2**.
`ay.emplace<T>(...)`	Destructs the object currently contained by **ay**; constructs a **T** in place, forwarding the arguments **...** to the appropriate constructor.
`ay.reset()`	Destroys the currently contained object.
`ay1.swap(ay2)` `swap(ay1, ay2)`	Swaps the objects contained by **ay1** and **ay2**.
`make_any<T>(...)`	Convenience function for constructing an any constructs a **T** in place, forwarding the arguments **...** to the appropriate constructor.
`t = any_cast<T>(ay)`	Casts **ay** into type **T**. Throws a `std::bad_any_cast` if the type **T** doesn't match the contained object's type.

variant

A *variant* is a class template that stores single values whose types are restricted to the user-defined list provided as template parameters. The variant is a type-safe `union` (refer to "Unions" on page 53). It shares a lot of functionality with the any type, but `variant` requires that you explicitly enumerate all the types that you'll store.

The stdlib has `std::variant` in the `<variant>` header, and Boost has `boost::variant` in the `<boost/variant.hpp>` header.

Listing 12-13 demonstrates creating another type called `BugblatterBeast` for `variant` to contain alongside `EscapeCapsule`.

```
#include <variant>

struct BugblatterBeast {
  BugblatterBeast() : is_ravenous{ true }, weight_kg{ 20000 } { }
  bool is_ravenous;
  int weight_kg; ❶
};
```

Listing 12-13: The `std::variant` can hold an object from one of a list of predefined types.

Aside from also containing a `weight_kg` member ❶, `BugblatterBeast` is totally independent from `EscapeCapsule`.

Constructing a variant

A `variant` can only be default constructed if one of two conditions is met:

- The first template parameter is default constructible.
- It is `monostate`, a type intended to communicate that a variant can have an empty state.

Because `BugblatterBeast` is default constructible (meaning it has a default constructor), make it the first type in the template parameter list so your variant is also default constructible, like so:

```
std::variant<BugblatterBeast, EscapeCapsule> hagunemnon;
```

To store a value into a `variant`, you use the `emplace` method template. As with `any`, a `variant` takes a single template parameter corresponding to the type you want to store. This template parameter must be contained in the list of template parameters for the `variant`. To extract a value, you use either of the non-member function templates `get` or `get_if`. These accept either the desired type or the index into the template parameter list corresponding to the desired type. If `get` fails, it throws a `bad_variant_access` exception, while `get_if` returns a `nullptr`.

You can determine which type corresponds with the current state of `variant` using the `index()` member, which returns the index of the current object's type within the template parameter list.

Listing 12-14 illustrates how to use `emplace` to change the state of a `variant` and `index` to determine the type of the contained object.

```
TEST_CASE("std::variant") {
  std::variant<BugblatterBeast, EscapeCapsule> hagunemnon;
  REQUIRE(hagunemnon.index() == 0); ❶

  hagunemnon.emplace<EscapeCapsule>(600); ❷
  REQUIRE(hagunemnon.index() == 1); ❸

  REQUIRE(std::get<EscapeCapsule>(hagunemnon).weight_kg == 600); ❹
  REQUIRE(std::get<1>(hagunemnon).weight_kg == 600); ❺
  REQUIRE_THROWS_AS(std::get<0>(hagunemnon), std::bad_variant_access); ❻
}
```

Listing 12-14: A `std::get` allows you to extract concrete types from `std::variant`.

After default constructing `hagunemnon`, invoking `index` yields 0 because this is the index of the correct template parameter ❶. Next, you emplace an EscapeCapsule ❷, which causes index to return 1 instead ❸. Both std::get<EscapeCapsule> ❹ and std::get<1> ❺ illustrate identical ways of extracting the contained type. Finally, attempting to invoke std::get to obtain a type that doesn't correspond with the current state of variant results in a bad_variant_access ❻.

You can use the non-member function std::visit to apply a callable object to a variant. This has the advantage of dispatching the correct function to handle whatever the contained object is without having to specify it explicitly with std::get. Listing 12-15 illustrates the basic usage.

```
TEST_CASE("std::variant") {
  std::variant<BugblatterBeast, EscapeCapsule> hagunemnon;
  hagunemnon.emplace<EscapeCapsule>(600); ❶
  auto lbs = std::visit([](auto& x) { return 2.2*x.weight_kg; }, hagunemnon); ❷
  REQUIRE(lbs == 1320); ❸
}
```

Listing 12-15: The std::visit *allows you to apply a callable object to a contained type of* std::variant*.*

First, you invoke emplace to store the value 600 into hagunemnon ❶. Because both BugblatterBeast and EscapeCapsule have a weight_kg member, you can use std::visit on hagunemnon with a lambda that performs the correct conversion (2.2 lbs per kg) to the weight_kg field ❷ and returns the result ❸ (notice that you don't have to include any type information).

Comparing variant and any

The universe is big enough to accommodate both any and variant. It's not possible to recommend one over the other generally, because each has its strengths and weaknesses.

An any is more flexible; it can take *any* type, whereas variant is only allowed to contain an object of a predetermined type. It also mostly avoids templates, so it's generally easier to program with.

A variant is less flexible, making it safer. Using the visit function, you can check for the safety of operations at compile time. With any, you would need to build your own visit-like functionality, and it would require runtime checking (for example, of the result of any_cast).

Finally, variant can be more performant than any. Although any is allowed to perform dynamic allocation if the contained type is too large, variant is not.

A Partial List of Supported Operations

Table 12-6 provides a list of the most supported std::variant operations. In this table, vt is a std::variant and t is an object of type T.

Table 12-6: The Most Supported `std::variant` Operations

Operation	Notes
`variant<...>{}`	Constructs an empty `variant` object. First template parameter must be default constructible.
`variant<...>{ vt }`	Copy constructs from **`vt`**.
`variant<...>{ move(vt) }`	Move constructs from **`vt`**.
`variant<...>{ move(t) }`	Constructs an `variant` object containing an in-place constructed object.
`vt = t`	Destructs the object currently contained by **`vt`**; copies **`t`**.
`vt = move(t)`	Destructs the object currently contained by **`vt`**; moves **`t`**.
`vt1 = vt2`	Copy assigns from **`vt2`**.
`vt1 = move(vt2)`	Move assigns from **`vt2`**.
`vt.emplace<T>(...)`	Destructs the object currently contained by **`vt`**; constructs a **`T`** in place, forwarding the arguments **`...`** to the appropriate constructor.
`vt.reset()`	Destroys the currently contained object.
`vt.index()`	Returns the zero-based index of the type of the currently contained object. (Order determined by template parameters of the `std::variant`.)
`vt1.swap(vt2)` `swap(vt1, vt2)`	Swaps the objects contained by **`vt1`** and **`vt2`**.
`make_variant<T>(...)`	Convenience function for constructing a `tuple`; constructs a **`T`** in place, forwarding the arguments **`...`** to the appropriate constructor.
`std::visit(vt, callable)`	Invokes **`callable`** with contained object.
`std::holds_alternative<T>(vt)`	Returns `true` if the contained object's type is **`T`**.
`std::get<I>(vt)` `std::get<T>(vt)`	Returns contained object if its type is **`T`** or the **`i`**th type. Otherwise, throws **`std::bad_variant_access`** exception.
`std::get_if<I>(&vt)` `std::get_if<T>(&vt)`	Returns a pointer to the contained object if its type is **`T`** or the **`i`**th type. Otherwise, returns `nullptr`.

`vt1 == vt2`
`vt1 != vt2`
`vt1 > vt2`
`vt1 >= vt2`
`vt1 < vt2`
`vt1 <= vt2`
	Compares the contained objects of **`vt1`** and **`vt2`**.

Date and Time

Between stdlib and Boost, a number of libraries are available that handle dates and times. When handling calendar dates and times, look to Boost's DateTime library. When you're trying get the current time or measure elapsed time, look to Boost's or stdlib's Chrono libraries and to Boost's Timer library.

Boost DateTime

Boost DateTime library supports date programming with a rich system based on the Gregorian calendar, which is the most widely used civil calendar internationally. Calendars are more complicated than they might seem at first glance. For example, consider the following excerpt from the US Naval Observatory's Introduction to Calendars, which describes the basics of leap years:

> Every year that is exactly divisible by four is a leap year, except for years that are exactly divisible by 100, but these centurial years are leap years if they are exactly divisible by 400. For example, the years 1700, 1800, and 1900 are not leap years, but the year 2000 is.

Rather than attempting to build your own solar calendar functions, just include DateTime's date-programming facilities with the following header:

```
#include <boost/date_time/gregorian/gregorian.hpp>
```

The principal type you'll use is the `boost::gregorian::date`, which is the primary interface for date-programming.

Constructing a date

Several options are available for constructing a `date`. You can default construct a `date`, which sets its value to the special date `boost::gregorian::not_a_date_time`. To construct a `date` with a valid date, you can use a constructor that accepts three arguments: a year, a month, and a date. The following statement constructs a `date d` with the date September 15, 1986:

```
boost::gregorian::date d{ 1986, 9, 15 };
```

Alternatively, you can construct a date from a string using the `boost::gregorian::from_string` utility function, like this:

```
auto d = boost::gregorian::from_string("1986/9/15");
```

If you pass an invalid date, the `date` constructor will throw an exception, such as `bad_year`, `bad_day_of_month`, or `bad_month`. For example, Listing 12-16 attempts to construct a date with September 32, 1986.

```
TEST_CASE("Invalid boost::Gregorian::dates throw exceptions") {
  using boost::gregorian::date;
  using boost::gregorian::bad_day_of_month;

  REQUIRE_THROWS_AS(date(1986, 9, 32), bad_day_of_month); ❶
}
```

Listing 12-16: The `boost::gregorian::date` constructor throws exceptions for bad dates.

Because September 32 isn't a valid day of the month, the `date` constructor throws a `bad_day_of_month` exception ❶.

NOTE *Due to a limitation in Catch, you cannot use braced initialization for* `date` *in the* `REQUIRE_THROWS_AS` *macro* ❶.

You can obtain the current day from the environment using the non-member function `boost::gregorian::day_clock::local_day` or `boost::gregorian::day_clock::universal_day` to obtain the local day based on the system's time zone settings and the UTC day, respectively:

```
auto d_local = boost::gregorian::day_clock::local_day();
auto d_univ = boost::gregorian::day_clock::universal_day();
```

Once you construct a date, you can't change its value (it's *immutable*). However, dates support copy construction and copy assignment.

Accessing Date Members

You can inspect the features of a `date` through its many `const` methods. Table 12-7 provides a partial list. In this table, `d` is a `date`.

Table 12-7: The Most Supported `boost::gregorian::date` Accessors

Accessor	Notes
`d.year()`	Returns the year portion of the `date`.
`d.month()`	Returns the month portion of the `date`.
`d.day()`	Returns the day portion of the `date`.
`d.day_of_week()`	Returns the day of the week as an `enum` of type `greg_day_of_week`.
`d.day_of_year()`	Returns the day of the year (from 1 to 366 inclusive).
`d.end_of_month()`	Returns a date object set to the last day of the month of `d`.
`d.is_not_a_date()`	Returns `true` if `d` is not a date.
`d.week_number()`	Returns the ISO 8601 week number.

Listing 12-17 illustrates how to construct a `date` and use the accessors in Table 12-7.

```
TEST_CASE("boost::gregorian::date supports basic calendar functions") {
  boost::gregorian::date d{ 1986, 9, 15 }; ❶
  REQUIRE(d.year() == 1986); ❷
  REQUIRE(d.month() == 9); ❸
  REQUIRE(d.day() == 15); ❹
  REQUIRE(d.day_of_year() == 258); ❺
  REQUIRE(d.day_of_week() == boost::date_time::Monday); ❻
}
```

Listing 12-17: The `boost::gregorian::date` *supports basic calendar functions.*

Here, you construct a `date` from September 15, 1986 ❶. From there, you extract the year ❷, month ❸, day ❹, day of the year ❺, and day of the week ❻.

Calendar Math

You can perform simple calendar math on dates. When you subtract one date from another, you get a `boost::gregorian::date_duration`. The main functionality of `date_duration` is storing an integral number of days, which you can extract using the `days` method. Listing 12-18 illustrates how to compute the number of days elapsed between two `date` objects.

```
TEST_CASE("boost::gregorian::date supports calendar arithmetic") {
  boost::gregorian::date d1{ 1986, 9, 15 }; ❶
  boost::gregorian::date d2{ 2019, 8, 1 }; ❷
  auto duration = d2 - d1; ❸
  REQUIRE(duration.days() == 12008); ❹
}
```

Listing 12-18: Subtracting `boost::gregorian::date` objects yields a `boost::gregorian::date_duration`.

Here, you construct a `date` for September 15, 1986 ❶ and for August 1, 2019 ❷. You subtract these two dates to yield a `date_duration` ❸. Using the `days` method, you can extract the number of days between the two dates ❹.

You can also construct a `date_duration` using a `long` argument corresponding to the number of days. You can add a `date_duration` to a date to obtain another date, as Listing 12-19 illustrates.

```
TEST_CASE("date and date_duration support addition") {
  boost::gregorian::date d1{ 1986, 9, 15 }; ❶
  boost::gregorian::date_duration dur{ 12008 }; ❷
  auto d2 = d1 + dur; ❸
  REQUIRE(d2 == boost::gregorian::from_string("2019/8/1")); ❹
}
```

Listing 12-19: Adding a `date_duration` to a `date` yields another `date`.

You construct a `date` for September 15, 1986 ❶ and 12,008 days for `duration` ❷. From Listing 12-18, you know that this day plus 12008 yields August 1, 2019. So after adding them ❸, the resulting day is as you expect ❹.

Date Periods

A *date period* represents the interval between two dates. DateTime provides a `boost::gregorian::date_period` class, which has three constructors, as described in Table 12-8. In this table, constructors `d1` and `d2` are `date` arguments and `dp` is a `date_period`.

Table 12-8: Supported `boost::gregorian::date_period` Constructors

Accessor	Notes
`date_period{ d1, d2 }`	Creates a period including **`d1`** but not **`d2`**; invalid if **`d2 <= d1`**.
`date_period{ d, n_days }`	Returns the month portion of the `date`.
`date_period{ dp }`	Copy constructor.

The date_period class supports many operations, such as the contain method, which takes a date argument and returns true if the argument is contained in the period. Listing 12-20 illustrates this operation.

```
TEST_CASE("boost::gregorian::date supports periods") {
  boost::gregorian::date d1{ 1986, 9, 15 }; ❶
  boost::gregorian::date d2{ 2019, 8, 1 }; ❷
  boost::gregorian::date_period p{ d1, d2 }; ❸
  REQUIRE(p.contains(boost::gregorian::date{ 1987, 10, 27 })); ❹
}
```

Listing 12-20: Using the contains method on a boost::gregorian::date_period to determine whether a date falls within a particular time interval

Here, you construct two dates, September 15, 1986 ❶ and August 1, 2019 ❷, which you use to construct a date_period ❸. Using the contains method, you can determine that the date_period contains the date October 27, 1987 ❹.

Table 12-9 contains a partial list of other date_period operations. In this table, p, p1, and p2 are date_period classes and d is a date.

Table 12-9: Supported boost::gregorian::date_period Operations

Accessor	Notes
p.begin()	Returns the first day.
p.last()	Returns the last day.
p.length()	Returns the number of days contained.
p.is_null()	Returns true if the period is invalid (for example, end is before start).
p.contains(d)	Returns true if d falls within p.
p1.contains(p2)	Returns true if all of p2 falls within p1.
p1.intersects(p2)	Returns true if any of p2 falls within p1.
p.is_after(d)	Returns true if p falls after d.
p.is_before(d)	Returns true if p falls before d.

Other DateTime Features

The Boost DateTime library contains three broad categories of programming:

Date Date programming is the calendar-based programming you just toured.

Time Time programming, which allows you to work with clocks with microsecond resolution, is available in the <boost/date_time/posix_time/posix_time.hpp> header. The mechanics are similar to date programming, but you work with clocks instead of Gregorian calendars.

Local-time Local-time programming is simply time-zone-aware time programming. It's available in the <boost/date_time/time_zone_base.hpp> header.

NOTE *For brevity, this chapter won't go into detail about time and local-time programming. See the Boost documentation for information and examples.*

Chrono

The stdlib Chrono library provides a variety of clocks in the `<chrono>` header. You typically use these when you need to program something that depends on time or for timing your code.

NOTE *Boost also offers a Chrono library in the* `<boost/chrono.hpp>` *header. It's a superset of stdlib's Chrono library, which includes, for example, process- and thread-specific clocks and user-defined output formats for time.*

Clocks

Three clocks are available in Chrono library; each provides a different guarantee, and all reside in the `std::chrono` namespace:

- The `std::chrono::system_clock` is the system-wide, real-time clock. It's sometimes also called the *wall clock*, the elapsed real time since an implementation-specific start date. Most implementations specify the Unix start date of January 1, 1970, at midnight.
- The `std::chrono::steady_clock` guarantees that its value will never decrease. This might seem absurd to guarantee, but measuring time is more complicated than it seems. For example, a system might have to contend with leap seconds or inaccurate clocks.
- The `std::chrono::high_resolution_clock` has the shortest *tick* period available: a tick is the smallest atomic change that the clock can measure.

Each of these three clocks supports the static member function `now`, which returns a time point corresponding to the current value of the clock.

Time Points

A *time point* represents a moment in time, and Chrono encodes time points using the `std::chrono::time_point` type. From a user perspective, `time_point` objects are very simple. They provide a `time_since_epoch` method that returns the amount of time elapsed between the time point and the clock's *epoch*. This elapsed time is called a *duration*.

An epoch is an implementation-defined reference time point denoting the beginning of a clock. The Unix Epoch (or POSIX time) begins on January 1, 1970, whereas the Windows Epoch begins on January 1, 1601 (corresponding with the beginning of a 400-year, Gregorian-calendar cycle).

The `time_since_epoch` method is not the only way to obtain a duration from a `time_point`. You can obtain the duration between two `time_point` objects by subtracting them.

Durations

A `std::chrono::duration` represents the time between the two `time_point` objects. Durations expose a `count` method, which returns the number of clock ticks in the duration.

Listing 12-21 shows how to obtain the current time from each of the three available clocks, extract the time since each clock's epoch as a duration, and then convert them to ticks.

```
TEST_CASE("std::chrono supports several clocks") {
  auto sys_now = std::chrono::system_clock::now(); ❶
  auto hires_now = std::chrono::high_resolution_clock::now(); ❷
  auto steady_now = std::chrono::steady_clock::now(); ❸


  REQUIRE(sys_now.time_since_epoch().count() > 0); ❹
  REQUIRE(hires_now.time_since_epoch().count() > 0); ❺
  REQUIRE(steady_now.time_since_epoch().count() > 0); ❻
}
```

Listing 12-21: The `std::chrono` supports several kinds of clocks.

You obtain the current time from the `system_clock` ❶, the `high_resolution_clock` ❷, and the `steady_clock` ❸. For each clock, you convert the time point into a `duration` since the clock's epoch using the `time_since_epoch` method. You immediately call `count` on the resulting duration to yield a tick count, which should be greater than zero ❹❺❻.

In addition to deriving durations from time points, you can construct them directly. The `std::chrono` namespace contains helper functions to generate durations. For convenience, Chrono offers a number of user-defined duration literals in the `std::literals::chrono_literals` namespace. These provide some syntactic sugar, convenient language syntax that makes life easier for the developer, for defining duration literals.

Table 12-10 shows the helper functions and their literal equivalents, where each expression corresponds to an hour's duration.

Table 12-10: `std::chrono` Helper Functions and User-Defined Literals for Creating Durations

Helper function	Literal equivalent
`nanoseconds(3600000000000)`	`3600000000000ns`
`microseconds(3600000000)`	`3600000000us`
`milliseconds(3600000)`	`3600000ms`
`seconds(3600)`	`3600s`
`minutes(60)`	`60m`
`hours(1)`	`1h`

For example, Listing 12-22 illustrates how to construct a duration of 1 second with `std::chrono::seconds` and another duration of 1,000 milliseconds using the `ms` duration literal.

```
#include <chrono>
TEST_CASE("std::chrono supports several units of measurement") {
  using namespace std::literals::chrono_literals; ❶
  auto one_s = std::chrono::seconds(1); ❷
  auto thousand_ms = 1000ms; ❸
  REQUIRE(one_s == thousand_ms); ❹
}
```

Listing 12-22: The `std::chrono` supports many units of measurement, which are comparable.

Here, you bring in the `std::literals::chrono_literals` namespace so you have access to the duration literals ❶. You construct a duration called `one_s` from the `seconds` helper function ❷ and another called `thousand_ms` from the `ms` duration literal ❸. These are equivalent because a second contains a thousand milliseconds ❹.

Chrono provides the function template `std::chrono::duration_cast` to cast a duration from one unit to another. As with other cast-related function templates, such as `static_cast`, `duration_cast` takes a single template parameter corresponding to the target duration and a single argument corresponding to the duration you want to cast.

Listing 12-23 illustrates how to cast a `nanosecond` duration into a `second` duration.

```
TEST_CASE("std::chrono supports duration_cast") {
  using namespace std::chrono; ❶
  auto billion_ns_as_s = duration_cast<seconds❷>(1'000'000'000ns❸);
  REQUIRE(billion_ns_as_s.count() == 1); ❹
}
```

Listing 12-23: The `std::chrono` supports `std::chrono::duration_cast`.

First, you bring in the `std::chrono` namespace for easy access to `duration_cast`, the duration helper functions, and the duration literals ❶. Next, you use the `ns` duration literal to specify a billion-nanosecond duration ❸, which you pass as the argument to `duration_cast`. You specify the template parameter of `duration_cast` as seconds ❷, so the resulting duration, `billion_ns_as_s`, equals 1 second ❹.

Waiting

Sometimes, you'll use durations to specify some period of time for your program to wait. The stdlib provides concurrency primitives in the `<thread>` header, which contains the non-member function `std::this_thread::sleep_for`. The `sleep_for` function accepts a `duration` argument corresponding to how long you want the current thread of execution to wait or "sleep."

Listing 12-24 shows how to employ `sleep_for`.

```
#include <thread>
#include <chrono>

TEST_CASE("std::chrono used to sleep") {
  using namespace std::literals::chrono_literals; ❶
  auto start = std::chrono::system_clock::now(); ❷
  std::this_thread::sleep_for(100ms); ❸
  auto end = std::chrono::system_clock::now(); ❹
  REQUIRE(end - start >= 100ms); ❺
}
```

Listing 12-24: The `std::chrono` works with `<thread>` to put the current thread to sleep.

As before, you bring in the `chrono_literals` namespace so you have access to the duration literals ❶. You record the current time according to `system_clock`, saving the resulting `time_point` into the `start` variable ❷. Next, you invoke `sleep_for` with a 100-millisecond duration (a tenth of a second) ❸. You then record the current time again, saving the resulting `time_point` into `end` ❹. Because the program slept for 100 milliseconds between calls to `std::chrono::system_clock`, the duration resulting from subtracting `start` from `end` should be at least `100ms` ❺.

Timing

To optimize code, you absolutely need accurate measurements. You can use Chrono to measure how long a series of operations takes. This enables you to establish that a particular code path is actually responsible for observed performance issues. It also enables you to establish an objective measure for the progress of your optimization efforts.

Boost's Timer library contains the `boost::timer::auto_cpu_timer` class in the `<boost/timer/timer.hpp>` header, which is an RAII object that begins timing in its constructor and stops timing in its destructor.

You can build your own makeshift `Stopwatch` class using just the stdlib Chrono library. The `Stopwatch` class can keep a reference to a `duration` object. In the `Stopwatch` destructor, you can set the `duration` via its reference. Listing 12-25 provides an implementation.

```
#include <chrono>

struct Stopwatch {
  Stopwatch(std::chrono::nanoseconds& result❶)
    : result{ result }, ❷
    start{ std::chrono::high_resolution_clock::now() } { } ❸
  ~Stopwatch() {
    result = std::chrono::high_resolution_clock::now() - start; ❹
  }
private:
  std::chrono::nanoseconds& result;
```

```
  const std::chrono::time_point<std::chrono::high_resolution_clock> start;
};
```

Listing 12-25: A simple `Stopwatch` class that computes the duration of its lifetime

The `Stopwatch` constructor requires a single `nanoseconds` reference ❶, which you store into the `result` field with a member initializer ❷. You also save the current time of the `high_resolution_clock` by setting the `start` field to the result of `now()` ❸. In the `Stopwatch` destructor, you again invoke `now()` on the `high_resolution_clock` and subtract `start` to obtain the duration of the lifetime of `Stopwatch`. You use the `result` reference to write the `duration` ❹.

Listing 12-26 shows the `Stopwatch` in action, performing a million floating-point divisions within a loop and computing the average time elapsed per iteration.

```
#include <cstdio>
#include <cstdint>
#include <chrono>

struct Stopwatch {
--snip--
};

int main() {
  const size_t n = 1'000'000; ❶
  std::chrono::nanoseconds elapsed; ❷
  {
    Stopwatch stopwatch{ elapsed }; ❸
    volatile double result{ 1.23e45 }; ❹
    for (double i = 1; i < n; i++) {
      result /= i; ❺
    }
  }
  auto time_per_division = elapsed.count() / double{ n }; ❻
  printf("Took %gns per division.", time_per_division); ❼
}
--------------------------------------------------------------------------
Took 6.49622ns per division. ❼
```

Listing 12-26: Using the `Stopwatch` to estimate the time taken for `double` division

First, you initialize a variable `n` to a million, which stores the total number of iterations your program will make ❶. You declare the `elapsed` variable, which will store the time elapsed across all the iterations ❷. Within a block, you declare a `Stopwatch` and pass an `elapsed` reference to the constructor ❸. Next, you declare a `double` called `result` with a junk value in it ❹. You declare this variable `volatile` so the compiler doesn't try to optimize the loop away. Within the loop, you do some arbitrary, floating-point division ❺.

Once the block completes, `stopwatch` destructs. This writes the duration of `stopwatch` to `elapsed`, which you use to compute the average number of nanoseconds per loop iteration and store into the `time_per_addition` variable ❻. You conclude the program by printing `time_per_division` with `printf` ❼.

Numerics

This section discusses handling numbers with a focus on common mathematical functions and constants; handling complex numbers; generating random numbers, numeric limits, and conversions; and computing ratios.

Numeric Functions

The stdlib Numerics and Boost Math libraries provide a profusion of numeric/mathematical functions. For the sake of brevity, this chapter presents only quick references. For detailed treatment, see [numerics] in the ISO C++ 17 Standard and the Boost Math documentation.

Table 12-11 provides a partial list of many common, non-member mathematical functions available in the stdlib's Math library.

Table 12-11: A Partial List of Common Math Functions in the stdlib

Function	Computes the . . .	Ints	Floats	Header
`abs(x)`	Absolute value of **x**.	✓		`<cstdlib>`
`div(x, y)`	Quotient and remainder of **x** divided by **y**.	✓		`<cstdlib>`
`abs(x)`	Absolute value of **x**.		✓	`<cmath>`
`fmod(x, y)`	Remainder of floating-point division of **x** by **y**.		✓	`<cmath>`
`remainder(x, y)`	Signed remainder of dividing **x** by **y**.	✓	✓	`<cmath>`
`fma(x, y, z)`	Multiply the first two arguments and add their product to the third argument; also called fused multiplication addition; that is, `x * y + z`.	✓	✓	`<cmath>`
`max(x, y)`	Maximum of **x** and **y**.	✓	✓	`<algorithm>`
`min(x, y)`	Minimum of **x** and **y**.	✓	✓	`<algorithm>`
`exp(x)`	Value of e^x.	✓	✓	`<cmath>`
`exp2(x)`	Value of 2^x.	✓	✓	`<cmath>`
`log(x)`	Natural log of x; that is, ln x.	✓	✓	`<cmath>`
`log10(x)`	Common log of x; that is, log10 x.	✓	✓	`<cmath>`
`log2(x)`	Base 2 log of x; that is, log10 x.	✓	✓	`<cmath>`
`gcd(x, y)`	Greatest common denominator of **x** and **y**.	✓		`<numeric>`
`lcm(x, y)`	Least common multiple of **x** and **y**.	✓		`<numeric>`
`erf(x)`	Gauss error function of **x**.	✓	✓	`<cmath>`
`pow(x, y)`	Value of x^y.	✓	✓	`<cmath>`
`sqrt(x)`	Square root of **x**.	✓	✓	`<cmath>`
`cbrt(x)`	Cube root of **x**.	✓	✓	`<cmath>`
`hypot(x, y)`	Square root of $x^2 + y^2$.	✓	✓	`<cmath>`

`sin(x)`
`cos(x)`
`tan(x)`
`asin(x)`
`acos(x)`
`atan(x)`
	Associated trigonometric function value.	✓	✓	`<cmath>`

Function	Computes the . . .	Ints	Floats	Header
sinh(**x**) cosh(**x**) tanh(**x**) asinh(**x**) acosh(**x**) atanh(**x**)	Associated hyperbolic function value.	✓	✓	<cmath>
ceil(**x**)	Nearest integer greater than or equal to **x**.	✓	✓	<cmath>
floor(**x**)	Nearest integer less than or equal to **x**.	✓	✓	<cmath>
round(**x**)	Nearest integer equal to **x**; rounds away from zero in midpoint cases.	✓	✓	<cmath>
isfinite(**x**)	Value true if **x** is a finite number.	✓	✓	<cmath>
isinf(**x**)	Value true if **x** is an infinite number.	✓	✓	<cmath>

NOTE *Other specialized mathematical functions are in the* <cmath> *header. For example, functions to compute Laguerre and Hermite polynomials, elliptic integrals, cylindrical Bessel and Neumann functions, and the Riemann zeta function appear in the header.*

Complex Numbers

A *complex number* is of the form a+bi, where i is an *imaginary number* that, when multiplied by itself, equals negative one; that is, i*i=-1. Imaginary numbers have applications in control theory, fluid dynamics, electrical engineering, signal analysis, number theory, and quantum physics, among other fields. The a portion of a complex number is called its *real component*, and the b portion is called the *imaginary component*.

The stdlib offers the std::complex class template in the <complex> header. It accepts a template parameter for the underlying type of the real and imaginary component. This template parameter must be one of the fundamental floating-point types.

To construct a complex, you can pass in two arguments: the real and the imaginary components. The complex class also supports copy construction and copy assignment.

The non-member functions std::real and std::imag can extract the real and imaginary components from a complex, respectively, as Listing 12-27 illustrates.

```
#include <complex>

TEST_CASE("std::complex has a real and imaginary component") {
  std::complex<double> a{0.5, 14.13}; ❶
  REQUIRE(std::real(a) == Approx(0.5)); ❷
  REQUIRE(std::imag(a) == Approx(14.13)); ❸
}
```

Listing 12-27: Constructing a std::complex *and extracting its components*

You construct a std::complex with a real component of 0.5 and an imaginary component of 14.13 ❶. You use std::real to extract the real component ❷ and std::imag to extract the imaginary component ❸.

Table 12-12 contains a partial list of supported operations with std::complex.

Table 12-12: A Partial List of std::complex Operations

Operation	Notes
c1+c2 c1-c2 c1*c2 c1/c2	Performs addition, subtraction, multiplication, and division.
c+s c-s c*s c/s	Converts the scalar s into a complex number with the real component equal to the scalar value and the imaginary component equal to zero. This conversion supports the corresponding complex operation (addition, subtraction, multiplication, or division) in the preceding row.
real(c)	Extracts real component.
imag(c)	Extracts imaginary component.
abs(c)	Computes magnitude.
arg(c)	Computes the phase angle.
norm(c)	Computes the squared magnitude.
conj(c)	Computes the complex conjugate.
proj(c)	Computes Riemann sphere projection.
sin(c)	Computes the sine.
cos(c)	Computes the cosine.
tan(c)	Computes the tangent.
asin(c)	Computes the arcsine.
acos(c)	Computes the arccosine.
atan(c)	Computes the arctangent.
c = polar(m, a)	Computes complex number determined by magnitude m and angle a.

Mathematical Constants

Boost offers a suite of commonly used mathematical constants in the <boost/math/constants/constants.hpp> header. More than 70 constants are available, and you can obtain them in float, double, or long double form by obtaining the relevant global variable from the boost::math::float_constants, boost::math::double_constants, and boost::math::long_double_constants respectively.

One of the many constants available is four_thirds_pi, which approximates $4\pi/3$. The formula for computing the volume of a sphere of radius r is $4\pi r^3/3$, so you could pull in this constant to make computing such a volume easy. Listing 12-28 illustrates how to compute the volume of a sphere with radius 10.

```
#include <cmath>
#include <boost/math/constants/constants.hpp>

TEST_CASE("boost::math offers constants") {
  using namespace boost::math::double_constants; ❶
  auto sphere_volume = four_thirds_pi * std::pow(10, 3); ❷
  REQUIRE(sphere_volume == Approx(4188.7902047));
}
```

Listing 12-28: The `boost::math` namespace offers constants

Here, you pull in the namespace `boost::math::double_constants`, which brings all the `double` versions of the Boost Math constants ❶. Next, you calculate the `sphere_volume` by computing `four_thirds_pi` times 10^3 ❷.

Table 12-13 provides some of the more commonly used constants in Boost Math.

Table 12-13: Some of the Most Common Boost Math Constants

Constant	Value	Approx.	Note
`half`	1/2	0.5	
`third`	1/3	0.333333	
`two_thirds`	2/3	0.66667	
`three_quarters`	3/4	0.75	
`root_two`	√2	1.41421	
`root_three`	√3	1.73205	
`half_root_two`	√2 / 2	0.707106	
`ln_two`	ln(2)	0.693147	
`ln_ten`	ln(10)	2.30258	
`pi`	π	3.14159	Archimedes' constant
`two_pi`	2π	6.28318	Circumference of unit circle
`four_thirds_pi`	4π/3	4.18879	Volume of unit sphere
`one_div_two_pi`	1/(2π)	1.59155	Gaussian integrals
`root_pi`	√π	1.77245	
`e`	e	2.71828	Euler's constant e
`e_pow_pi`	e^{π}	23.14069	Gelfond's constant
`root_e`	√e	1.64872	
`log10_e`	log10(e)	0.434294	
`degree`	π / 180	0.017453	Number of radians per degree
`radian`	180 / π	57.2957	Number of degrees per radian
`sin_one`	sin(1)	0.84147	
`cos_one`	cos(1)	0.5403	
`phi`	(1 + √5) / 2	1.61803	Phidias' golden ratio φ
`ln_phi`	ln(φ)	0.48121	

Random Numbers

In some settings, it's often necessary to generate random numbers. In scientific computing, you might need to run large numbers of simulations based on random numbers. Such numbers need to emulate draws from random processes with certain characteristics, such as coming from a Poisson or normal distribution. In addition, you usually want these simulations to be repeatable, so the code responsible for generating randomness—the *random number engine*—should produce the same output given the same input. Such random number engines are sometimes called *pseudo*-random number engines.

In cryptography, you might require random numbers to instead secure information. In such settings, it must be virtually impossible for someone to obtain a similar stream of random numbers; so accidental use of pseudo-random number engines often seriously compromises an otherwise secure cryptosystem.

For these reasons and others, *you should never attempt to build your own random number generator.* Building a correct random number generator is surprisingly difficult. It's too easy to introduce patterns into your random number generator, which can have nasty and hard to diagnose side effects on systems that use your random numbers as input.

NOTE *If you're interested in random number generation, refer to Chapter 2 of* Stochastic Simulation *by Brian D. Ripley for scientific applications and Chapter 2 of* Serious Cryptography *by Jean-Philippe Aumasson for cryptographic applications.*

If you're in the market for random numbers, look no further than the Random libraries available in the stdlib in the `<random>` header or in Boost in the `<boost/math/...>` headers.

Random Number Engines

Random number engines generate random bits. Between Boost and stdlib, there is a dizzying array of candidates. Here's a general rule: if you need repeatable pseudo-random numbers, consider using the Mersenne Twister engine `std::mtt19937_64`. If you need cryptographically secure random numbers, consider using `std::random_device`.

The Mersenne Twister has some desirable statistical properties for simulations. You provide its constructor with an integer seed value, which completely determines the sequence of random numbers. All random engines are function objects; to obtain a random number, use the function call `operator()`. Listing 12-29 shows how to construct a Mersenne Twister engine with the seed 91586 and invoke the resulting engine three times.

```
#include <random>
TEST_CASE("mt19937_64 is pseudorandom") {
  std::mt19937_64 mt_engine{ 91586 }; ❶
```

```
  REQUIRE(mt_engine() == 8346843996631475880); ❷
  REQUIRE(mt_engine() == 2237671392849523263); ❸
  REQUIRE(mt_engine() == 7333164488732543658); ❹
}
```

Listing 12-29: The `mt19937_64` is a pseudo-random number engine.

Here, you construct an `mt19937_64` Mersenne Twister engine with the seed 91586 ❶. Because it's a pseudo-random engine, you're guaranteed to get the same sequence of random numbers ❷❸❹ each time. This sequence is determined entirely by the seed.

Listing 12-30 illustrates how to construct a `random_device` and invoke it to obtain a cryptographically secure random value.

```
TEST_CASE("std::random_device is invocable") {
  std::random_device rd_engine{}; ❶
  REQUIRE_NOTHROW(rd_engine()); ❷
}
```

Listing 12-30: The `random_device` is a function object.

You construct a `random_device` using the default constructor ❶. The resulting object `rd_engine` ❷ is invokable, but you should treat the object as opaque. Unlike the Mersenne Twister in Listing 12-29, `random_device` is unpredictable by design.

NOTE *Because computers are deterministic by design, the `std::random_device` cannot make any strong guarantees about cryptographic security.*

Random Number Distributions

A *random number distribution* is a mathematical function that maps a number to a probability density. Roughly, the idea is that if you take infinite samples from a random variable that has a particular distribution and you plot the relative frequencies of your sample values, that plot would look like the distribution.

Distributions break out into two broad categories: *discrete* and *continuous*. A simple analogy is that discrete distributions map integral values, and continuous distributions map floating-point values.

Most distributions accept customization parameters. For example, the normal distribution is a continuous distribution that accepts two parameters: a mean and a variance. Its density has a familiar bell shape centered around the mean, as shown in Figure 12-1. The discrete uniform distribution is a random number distribution that assigns equal probability to the numbers between some minimum and maximum. Its density looks perfectly flat across its range from minimum to maximum, as shown in Figure 12-2.

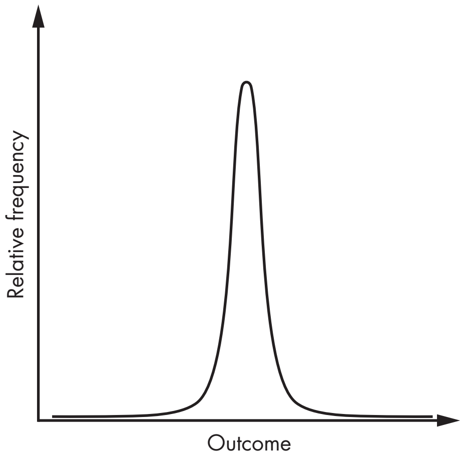


Figure 12-1: A representation of the normal distribution's probability density function

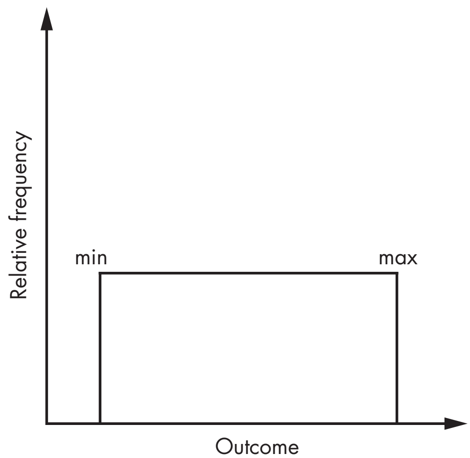


Figure 12-2: A representation of the uniform distribution's probability density function

You can easily generate random numbers from common statistical distributions, such as the uniform and the normal, using the same stdlib Random library. Each distribution accepts some parameters in its constructor, corresponding to the underlying distribution's parameters. To draw a random variable from the distribution, you use the function call `operator()` and pass in an instance of a random number engine, such as a Mersenne Twister.

The `std::uniform_int_distribution` is a class template available in the `<random>` header that takes a single template parameter corresponding to the type you want returned by draws from the distribution, like an `int`. You specify the uniform distribution's minimum and maximum by passing them in as constructor parameters. Each number in the range has equal probability. It's perhaps the most common distribution to arise in general software engineering contexts.

Listing 12-31 illustrates how to take a million draws from a uniform distribution with a minimum of 1 and a maximum of 10 and compute the sample mean.

```
TEST_CASE("std::uniform_int_distribution produces uniform ints") {
  std::mt19937_64 mt_engine{ 102787 }; ❶
  std::uniform_int_distribution<int> int_d{ 0, 10 }; ❷
  const size_t n{ 1'000'000 }; ❸
  int sum{}; ❹
  for (size_t i{}; i < n; i++)
    sum += int_d(mt_engine); ❺
  const auto sample_mean = sum / double{ n }; ❻
  REQUIRE(sample_mean == Approx(5).epsilon(.1)); ❼
}
```

Listing 12-31: The `uniform_int_distribution` simulates draws from the discrete uniform distribution.

You construct a Mersenne Twister with the seed 102787 ❶ and then construct a uniform_int_distribution with a minimum of 0 and a maximum of 10 ❷. Then you initialize a variable n to hold the number of iterations ❸ and initialize a variable to hold the sum of all the uniform random variables ❹. In the loop, you draw random variables from the uniform distribution with operator(), passing in the Mersenne Twister instance ❺.

The mean of a discrete uniform distribution is the minimum plus the maximum divided by 2. Here, int_d has a mean of 5. You can compute a sample mean by dividing sum by the number of samples n ❻. With high confidence, you assert that this sample_mean is approximately 5 ❼.

A Partial List of Random Number Distributions

Table 12-14 contains a partial list of the random number distributions in <random>, their default template parameters, and their constructor parameters.

Table 12-14: Random Number Distributions in <random>

Distribution	Notes
uniform_int_distribution<int>{ **min, max** }	Discrete uniform distribution with minimum **min** and maximum **max**.
uniform_real_distribution<double>{ **min, max** }	Continuous uniform distribution with minimum **min** and maximum **max**.
normal_distribution<double>{ **m, s** }	Normal distribution with mean **m** and standard deviation **s**. Commonly used to model the additive product of many independent random variables. Also called the Gaussian distribution.
lognormal_distribution<double>{ **m, s** }	Log-normal distribution with mean **m** and standard deviation **s**. Commonly used to model the multiplicative product of many independent random variables. Also called Galton's distribution.
chi_squared_distribution<double>{ **n** }	Chi-squared distribution with degrees of freedom **n**. Commonly used in inferential statistics.
cauchy_distribution<double>{ **a, b** }	Cauchy distribution with location parameter **a** and scale parameter **b**. Used in physics. Also called the Lorentz distribution.
fisher_f_distribution<double>{ **m, n** }	F distribution with degrees of freedom **m** and **n**. Commonly used in inferential statistics. Also called the Snedecor distribution.
student_t_distribution<double>{ **n** }	T distribution with degrees of freedom **n**. Commonly used in inferential statistics. Also called the Student's T distribution.

(continued)

Table 12-14: Random Number Distributions in `<random>` (continued)

Distribution	Notes
`bernoulli_distribution{ p }`	Bernoulli distribution with success probability **p**. Commonly used to model the result of a single, Boolean-valued outcome.
`binomial_distribution<int>{ n, p }`	Binomial distribution with **n** trials and success probability **p**. Commonly used to model the number of successes when sampling with replacement in a series of Bernoulli experiments.
`geometric_distribution<int>{ p }`	Geometric distribution with success probability **p**. Commonly used to model the number of failures occurring before the first success in a series of Bernoulli experiments.
`poisson_distribution<int>{ m }`	Poisson distribution with mean **m**. Commonly used to model the number of events occurring in a fixed interval of time.
`exponential_distribution<double>{ l }`	Exponential distribution with mean 1/**l**, where **l** is known as the lambda parameter. Commonly used to model the amount of time between events in a Poisson process.
`gamma_distribution<double>{ a, b }`	Gamma distribution with shape parameter **a** and scale parameter **b**. Generalization of the exponential distribution and chi-squared distribution.
`weibull_distribution<double>{ k, l }`	Weibull distribution with shape parameter **k** and scale parameter **l**. Commonly used to model time to failure.
`extreme_value_distribution<double>{ a, b }`	Extreme value distribution with location parameter **a** and scale parameter **b**. Commonly used to model maxima of independent random variables. Also called the Gumbel type-I distribution.

NOTE *Boost Math offers more random number distributions in the `<boost/math/...>` series of headers, for example, the beta, hypergeometric, logistic, and inverse normal distributions.*

Numeric Limits

The stdlib offers the class template `std::numeric_limits` in the `<limits>` header to provide you with compile time information about various properties for arithmetic types. For example, if you want to identify the smallest finite value for a given type `T`, you can use the static member function `std::numeric_limits<T>::min()` to obtain it.

Listing 12-32 illustrates how to use `min` to facilitate an underflow.

```
#include <limits>
TEST_CASE("std::numeric_limits::min provides the smallest finite value.") {
  auto my_cup = std::numeric_limits<int>::min(); ❶
  auto underfloweth = my_cup - 1; ❷
  REQUIRE(my_cup < underfloweth); ❸
}
```

Listing 12-32: Using `std::numeric_limits<T>::min()` to facilitate an `int` underflow. Although at press time the major compilers produce code that passes the test, this program contains undefined behavior.

First, you set the `my_cup` variable equal to the smallest possible `int` value by using `std::numeric_limits<int>::min()` ❶. Next, you intentionally cause an underflow by subtracting 1 from `my_cup` ❷. Because `my_cup` is the minimum value an `int` can take, `my_cup` runneth under, as the saying goes. This causes the deranged situation that `underfloweth` is greater than `my_cup` ❸, even though you initialized `underfloweth` by subtracting from `my_cup`.

NOTE *Such silent underflows have been the cause of untold numbers of software security vulnerabilities. Don't rely on this undefined behavior!*

Many static member functions and member constants are available on `std::numeric_limits`. Table 12-15 lists some of the most common.

Table 12-15: Some Common Member Constants in `std::numeric_limits`

Operation	Notes
`numeric_limits<T>::is_signed`	`true` if `T` is signed.
`numeric_limits<T>::is_integer`	`true` if `T` is an integer.
`numeric_limits<T>::has_infinity`	Identifies whether `T` can encode an infinite value. (Usually, all floating-point types have an infinite value, whereas integral types don't.)
`numeric_limits<T>::digits10`	Identifies the number of digits `T` can represent.
`numeric_limits<T>::min()`	Returns the smallest value of `T`.
`numeric_limits<T>::max()`	Returns the largest value of `T`.

NOTE *Boost Integer provides some additional facilities for introspecting integer types, such as determining the fastest or smallest integer, or the smallest integer with at least* N *bits.*

Boost Numeric Conversion

Boost provides the Numeric Conversion library, which contains a collection of tools to convert between numeric objects. The `boost::converter` class template in the `<boost/numeric/conversion/converter.hpp>` header encapsulates code to perform a specific numeric conversion from one type to another. You must provide two template parameters: the target type `T` and the source type `S`. You can specify a numeric converter that takes a `double` and converts it to an `int` with the simple type alias `double_to_int`:

```
#include <boost/numeric/conversion/converter.hpp>
using double_to_int = boost::numeric::converter<int❶, double❷>;
```

To convert with your new type alias `double_to_int`, you have several options. First, you can use its static method `convert`, which accepts a `double` ❷ and returns an `int` ❶, as Listing 12-33 illustrates.

```
TEST_CASE("boost::converter offers the static method convert") {
  REQUIRE(double_to_int::convert(3.14159) == 3);
}
```

Listing 12-33: The `boost::converter` *offers the static method* `convert`*.*

Here, you simply invoke the `convert` method with the value 3.14159, which `boost::convert` converts to 3.

Because `boost::convert` provides the function call `operator()`, you can construct a function object `double_to_int` and use it to convert, as in Listing 12-34.

```
TEST_CASE("boost::numeric::converter implements operator()") {
  double_to_int dti; ❶
  REQUIRE(dti(3.14159) == 3); ❷
  REQUIRE(double_to_int{}(3.14159) == 3); ❸
}
```

Listing 12-34: The `boost::converter` *implements* `operator()`*.*

You construct a `double_to_int` function object called `dti` ❶, which you invoke with the same argument, 3.14159 ❷, as in Listing 12-33. The result is the same. You also have the option of constructing a temporary function object and using `operator()` directly, which yields identical results ❸.

A major advantage of using `boost::converter` instead of alternatives like `static_cast` is runtime bounds checking. If a conversion would cause an overflow, `boost::converter` will throw a `boost::numeric::positive_overflow` or `boost::numeric::negative_overflow`. Listing 12-35 illustrates this behavior when you attempt to convert a very large `double` into an `int`.

```
#include <limits>
TEST_CASE("boost::numeric::converter checks for overflow") {
  auto yuge = std::numeric_limits<double>::max(); ❶
  double_to_int dti; ❷
  REQUIRE_THROWS_AS(dti(yuge)❸, boost::numeric::positive_overflow❹);
}
```

Listing 12-35: The `boost::converter` *checks for overflow.*

You use numeric_limits to obtain a yuge value ❶. You construct a double_to_int converter ❷, which you use to attempt a conversion of yuge to an int ❸. This throws a positive_overflow exception because the value is too large to store ❹.

It's possible to customize the conversion behavior of boost::converter using template parameters. For example, you can customize the overflow handling to throw a custom exception or perform some other operation. You can also customize rounding behavior so that rather than truncating off the decimal from a floating-point value, you perform custom rounding. See the Boost Numeric Conversion documentation for details.

If you're happy with the default boost::converter behavior, you can use the boost::numeric_cast function template as a shortcut. This function template accepts a single template parameter corresponding to the target type of the conversion and a single argument corresponding to the source number. Listing 12-36 provides an update to Listing 12-35 that uses boost::numeric_cast instead.

```
#include <limits>
#include <boost/numeric/conversion/cast.hpp>

TEST_CASE("boost::boost::numeric_cast checks overflow") {
  auto yuge = std::numeric_limits<double>::max(); ❶
  REQUIRE_THROWS_AS(boost::numeric_cast<int>(yuge), ❷
                    boost::numeric::positive_overflow ❸);
}
```

Listing 12-36: The boost::numeric_cast function template also performs runtime bounds checking.

As before, you use numeric_limits to obtain a yuge value ❶. When you try to numeric_cast yuge into an int ❷, you get a positive_overflow exception because the value is too large to store ❸.

NOTE *The boost::numeric_cast function template is a suitable replacement for the narrow_cast you hand-rolled in Listing 6-6 on page 154.*

Compile-Time Rational Arithmetic

The stdlib std::ratio in the <ratio> header is a class template that enables you to compute rational arithmetic at compile time. You provide two template parameters to std::ratio: a numerator and a denominator. This defines a new type that you can use to compute rational expressions.

The way you perform compile-time computation with std::ratio is by using template metaprogramming techniques. For example, to multiply two ratio types, you can use the std::ratio_multiply type, which takes the two ratio types as template parameters. You can extract the numerator and denominator of the result using static member variables on the resulting type.