Marius Bancila

Modern C++ Programming Cookbook

Over 100 recipes to help you overcome your difficulties with C++ programming and gain a deeper understanding of the working of modern C++

Packt

Title Page

Modern C++ Programming Cookbook

Over 100 recipes to help you overcome your difficulties with C++ programming and gain a deeper understanding of the working of modern C++

Marius Bancila

BIRMINGHAM - MUMBAI

Copyright

Modern C++ Programming Cookbook

First published: May 2017

Production reference: 1090517

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78646-518-4

www.packtpub.com

Credits

Author Marius Bancila	**Copy Editor** Dhanya Baburaj
Reviewer David V. Corbin	**Project Coordinator** Vaidehi Sawant
Commissioning Editor Aaron Lazar	**Proofreader** Safis Editing
Acquisition Editor Nitin Dasan	**Indexer** Aishwarya Gangawane

Content Development Editor Anurag Ghogre	**Cover Work** Arvindkumar Gupta
Technical Editor Subhalaxmi Nadar	**Production Coordinator** Arvindkumar Gupta

About the Author

Marius Bancila is a software engineer with 14 years of experience in developing solutions for the industrial and financial sectors. He focuses on Microsoft technologies and mainly develops desktop applications with C++ and C#. Over the years, he has worked with other languages and technologies including Java, HTML/CSS, PHP, and JavaScript.

Marius is passionate about sharing his technical expertise with others, and for that reason, he has been recognized as a Microsoft MVP for more than a decade. He has been an active contributor to forums and other developer communities where he has published many articles, for which he has won multiple awards. He also created and contributed to several open source libraries. He is a cofounder of Codexpert, a Romanian community for C++ developers. He is based in Timisoara, Romania, and works as a system architect, building accounting and logistic solutions for a major European software vendor. He can be followed on Twitter at https://twitter.com/mariusbancila.

I would like to thank Packt Publishing for getting me on board with this wonderful project that I greatly enjoyed. Many thanks to Anurag Ghogre, Subhalaxmi Nadar and Nitin Dasan for the constant support shown throughout the project, as well as the other members of the team involved. A special thanks to David Corbin who provided valuable feedback to make this book better. Last, but not least, a big thank you to my wife who has been very patient and supportive through the many days and nights I spent writing this book.

About the Reviewer

David V. Corbin began programming during the heyday of the mini-computer era, starting with the DEC PDP-8. His early career was in the defense industry, progressing from the company's first software technician to being the technical lead of the engineering software department, with much of the work being done in C. He cofounded Dynamic Concepts in 1984 to facilitate the introduction of the PC into business environments (this is the same year the original IBM AT was introduced).

By the early 1990s, much of his application development had started migrating to C++. Even after 25 years, C++ remains a valued tool in his development arsenal. In 2005, he began to focus on improving the software development and delivery process via the application of ALM principles. Today, he continues as the President and Chief Architect of Dynamic Concepts and works directly with clients, providing guidance in the rapidly changing ecosystem.

I would like to thank Packt Publishing for the opportunity to be a technical reviewer of this book. I have known Marius Bancila for a decade and he is one of the brightest developers I have known. His work is sure to have a positive impact on the entire C++ developer community in their quest to become familiar with the modern elements of C++.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

https://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at https://www.amazon.com/dp/1786465183.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

C++ is one of the most popular and most widely used programming languages, and it has been like that for three decades. Designed with a focus on performance, efficiency, and flexibility, C++ combines paradigms such as object-oriented, imperative, generic, and, more recently, functional programming. C++ is standardized by the International Organization for Standardization (ISO) and has undergone massive changes over the last decade. With the standardization of C++11, the language has entered into a new age, which has been widely referred to as modern C++. Type inference, move semantics, lambda expression, smart pointers, uniform initialization, variadic templates, and many other recent features have changed the way we write code in C++ to the point that it almost looks like a new programming language.

This book addresses many of the new features included in C++11, C++14, and the forthcoming C++17. This book is organized in recipes, each covering one particular language or library feature, or a common problem developers face and its typical solution using modern C++. Through more than 100 recipes, you will learn to master both core language features and the standard libraries, including those for strings, containers, algorithms, iterators, input/output, regular expressions, threads, filesystem, atomic operations, and utilities.

This book took about 6 months to write, and during this time the work on the C++17 standard has progressed. At the point of writing this preface, the standard is completed, but is yet to be approved and published later this year. A number of recipes in this book cover C++17 features, including fold expressions, constexpr if, structured bindings, new standard attributes, optional, any, variant and string_view types, and the filesystem library.

All the recipes in the book contain code samples that show how to use a feature or how to solve a problem. These code samples have been written using Visual Studio 2017, but have been also compiled using Clang and GCC. Since the support for various language and library features have been gradually added to all these compilers, it is recommended that you use the latest version to ensure that all of them are supported. At the time of writing this, the latest versions are GCC 7.0, Clang 5.0, and VC++ 2017 (version 19.1). Although GCC and Clang support all the features addressed in this book, VC++ is yet to support fold expressions, constexpr if, and searchers for std::search().

What this book covers

Chapter 1, *Learning Modern Core Language Features*, teaches you about modern core language features including type inference, uniform initialization, scoped enums, range-based for loops, structured bindings, and others.

Chapter 2, *Working with Numbers and Strings*, discusses how to convert between numbers and strings, generate pseudo-random numbers, work with regular expressions, and various types of string.

Chapter 3, *Exploring Functions*, dives into defaulted and deleted functions, variadic templates, lambda expressions, and higher-order functions.

Chapter 4, *Preprocessor and Compilation*, takes a look at various aspects of compilation, from how to perform conditional compilation, to compile-time assertions, code generation, or hinting the compiler with attributes.

Chapter 5, *Standard Library Containers, Algorithms, and Iterators*, introduces you to several standard containers, many algorithms, and teaches you how to write your own random access iterator.

Chapter 6, *General Purpose Utilities*, dives into the chrono library; the any, optional, and variant types; and learn about type traits.

Chapter 7, *Working with Files and Streams*, explains how to read and write data to/from streams, use I/O manipulators to control streams, and explores the filesystem library.

Chapter 8, *Leveraging Threading and Concurrency*, informs you how to work with threads, mutexes, locks, condition variables, promises, futures, and atomic types.

Chapter 9, *Robustness and Performance*, focuses on exceptions, constant correctness, type casts, smart pointers, and move semantics.

Chapter 10, *Implementing Patterns and Idioms*, covers various useful patterns and idioms, such as the pimpl idiom, the non-virtual interface idiom, or the curiously recurring template pattern.

Chapter 11, *Exploring Testing Frameworks*, helps you get a kickstart with three of the most widely used testing frameworks, Boost.Test, Google Test, and Catch.

What you need for this book

The code presented in the book is available for download from your account at https://www.packtpub.com/, although I encourage you to try writing all the samples by yourself. In order to compile them, you need VC++ 2017 on Windows and GCC 7.0 or Clang 5.0 on Linux and Mac. If you don’t have the latest version of the compiler, or you want to try another compiler, you can use one that is available online. Although there are various online platforms that you could use, I recommend https://wandbox.org/ for GCC and Clang and http://webcompiler.cloudapp.net/ for VC++.

Who this book is for

This book is intended for all C++ developers, regardless of their experience level. The typical reader is an entry- or medium-level C++ developer who wants to master the language and become a prolific modern C++ developer. The experienced C++ developer will find a good reference for many C++11, C++14, and C++17 language and library features that may come in handy from time to time. The book consists of more than one hundred recipes that are simple, intermediate, or advanced. However, they all require prior knowledge of C++, and that includes functions, classes, templates, namespaces, macros, and others. Therefore, if you are not familiar with the language, it is recommended that you first read an introductory book to familiarize yourself with the core aspects, and then proceed with this book.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it, How it works, There’s more, and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There’s more…

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, folder names, filenames, file extensions, path names, URLs, user input, and others are shown as follows: `#include <iostream>`.

A block of code is set as follows:

```
#include <iostream>

int main()
{
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

You can also download the code files by clicking on the Code Files button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the Search box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Modern-Cpp-Programming-Cookbook. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

Learning Modern Core Language Features

The recipes included in this chapter are as follows:

- Using auto whenever possible
- Creating type aliases and alias templates
- Understanding uniform initialization
- Understanding the various forms of non-static member initialization
- Controlling and querying object alignment
- Using scoped enumerations
- Using override and final for virtual methods
- Using range-based for loops to iterate on a range
- Enabling range-based for loops for custom types
- Using explicit constructors and conversion operators to avoid implicit conversion
- Using unnamed namespaces instead of static globals
- Using inline namespaces for symbol versioning
- Using structured bindings to handle multi-return values

Introduction

The C++ language has gone through a major transformation in the past decade with the development and release of C++11 and then later with its newer versions C++14 and C++17. These new standards have introduced new concepts, simplified or extended existing syntax and semantics, and overall transformed the way we write code. C++11 looks like a new language, and code written using the new standards is called modern C++ code.

Using auto whenever possible

Automatic type deduction is one of the most important and widely used features in modern C++. The new C++ standards have made it possible to use `auto` as a placeholder for types in various contexts and let the compiler deduce the actual type. In C++11, `auto` can be used for declaring local variables and for the return type of a function with a trailing return type. In C++14, `auto` can be used for the return type of a function without specifying a trailing type and for parameter declarations in lambda expressions. Future standard versions are likely to expand the use of `auto` to even more cases. The use of `auto` in these contexts has several important benefits. Developers should be aware of them, and prefer `auto` whenever possible. An actual term was coined for this by Andrei Alexandrescu and promoted by Herb Sutter—*almost always auto* (*AAA*).

How to do it…

Consider using `auto` as a placeholder for the actual type in the following situations:

- To declare local variables with the form `auto name = expression` when you do not want to commit to a specific type:

```
auto i = 42;          // int
auto d = 42.5;        // double
auto s = "text";      // char const *
auto v = { 1, 2, 3 }; // std::initializer_list<int>
```

- To declare local variables with the `auto name = type-id { expression }` form when you need to commit to a specific type:

```
auto b  = new char[10]{ 0 };            // char*
auto s1 = std::string {"text"};         // std::string
auto v1 = std::vector<int> { 1, 2, 3 }; // std::vector<int>
auto p  = std::make_shared<int>(42);    // std::shared_ptr<int>
```

- To declare named lambda functions, with the form `auto name = lambda-expression`, unless the lambda needs to be passed or return to a function:

```
auto upper = [](char const c) {return toupper(c); };
```

- To declare lambda parameters and return values:

```
auto add = [](auto const a, auto const b) {return a + b;};
```

- To declare function return type when you don’t want to commit to a specific type:

```
template <typename F, typename T>
auto apply(F&& f, T value)
{
  return f(value);
}
```

How it works…

The `auto` specifier is basically a placeholder for an actual type. When using `auto`, the compiler deduces the actual type from the following instances:

- From the type of the expression used to initialize a variable, when `auto` is used to declare variables.
- From the trailing return type or the type of the return expression of a function, when `auto` is used as a placeholder for the return type of a function.

In some cases, it is necessary to commit to a specific type. For instance, in the preceding example, the compiler deduces the type of `s` to be `char const *`. If the intention was to have a `std::string`, then the type must be specified explicitly. Similarly, the type of `v` was deduced as `std::initializer_list<int>`. However, the intention could be to have a `std::vector<int>`. In such cases, the type must be specified explicitly on the right side of the assignment.

There are some important benefits of using the auto specifier instead of actual types; the following is a list of, perhaps, the most important ones:

- It is not possible to leave a variable uninitialized. This is a common mistake that developers do when declaring variables specifying the actual type, but it is not possible with `auto` that requires an initialization of the variable in order to deduce the type.
- Using `auto` ensures that you always use the correct type and that implicit conversion will not occur. Consider the following example where we retrieve the size of a vector to a local variable. In the first case, the type of the variable is `int`, though the `size()` method returns `size_t`. That means an implicit conversion from `size_t` to `int` will occur. However, using `auto` for the type will deduce the correct type, that is, `size_t`:

```
auto v = std::vector<int>{ 1, 2, 3 };
int size1 = v.size();
// implicit conversion, possible loss of data
auto size2 = v.size();
auto size3 = int{ v.size() };  // error, narrowing conversion
```

- Using `auto` promotes good object-oriented practices, such as preferring interfaces over implementations. The lesser the number of types specified the more generic the code is and more open to future changes, which is a fundamental principle of object-oriented programming.
- It means less typing and less concern for actual types that we don’t care about anyways. It is very often that even though we explicitly specify the type, we don’t actually care about it. A very common case is with iterators, but one can think of many more. When you want to iterate over a range, you don’t care about the actual type of the iterator. You are only interested in the iterator itself; so, using `auto` saves time used for typing possibly long names and helps you focus on actual code and not type names. In the following example, in the first `for` loop, we explicitly use the type

of the iterator. It is a lot of text to type, the long statements can actually make the code less readable, and you also need to know the type name that you actually don't care about. The second loop with the `auto` specifier looks simpler and saves you from typing and caring about actual types.

```
std::map<int, std::string> m;
for (std::map<int,std::string>::const_iterator it = m.cbegin();
  it != m.cend(); ++it)
{ /*...*/ }

for (auto it = m.cbegin(); it != m.cend(); ++it)
{ /*...*/ }
```

- Declaring variables with `auto` provides a consistent coding style with the type always in the right-hand side. If you allocate objects dynamically, you need to write the type both on the left and right side of the assignment, for example, `int p = new int(42)`. With `auto`, the type is specified only once on the right side.

However, there are some gotchas when using `auto`:

- The `auto` specifier is only a placeholder for the type, not for the `const`/`volatile` and references specifiers. If you need a `const`/`volatile` and/or reference type, then you need to specify them explicitly. In the following example, `foo.get()` returns a reference to `int`; when variable `x` is initialized from the return value, the type deduced by the compiler is `int`, and not `int&`. Therefore, any change to `x` will not propagate to `foo.x_`. In order to do so, one should use `auto&`:

```
class foo {
  int x_;
public:
  foo(int const x = 0) :x_{ x } {}
  int& get() { return x_; }
};

foo f(42);
auto x = f.get();
x = 100;
std::cout << f.get() << std::endl; // prints 42
```

- It is not possible to use `auto` for types that are not moveable:

```
auto ai = std::atomic<int>(42); // error
```

- It is not possible to use auto for multi-word types, such as `long long`, `long double`, or `struct foo`. However, in the first case, the possible workarounds are to use literals or type aliases; as for the second, using `struct`/`class` in that form is only supported in C++ for C compatibility and should be avoided anyways:

```
auto l1 = long long{ 42 }; // error
auto l2 = llong{ 42 };     // OK
auto l3 = 42LL;            // OK
```

- If you use the `auto` specifier but still need to know the type, you can do so in any IDE by putting the cursor over a variable, for instance. If you leave the IDE, however, that is not possible anymore, and the only way to know the actual type is to deduce it

yourself from the initialization expression, which could probably mean searching through the code for function return types.

The `auto` can be used to specify the return type from a function. In C++11, this requires a trailing return type in the function declaration. In C++14, this has been relaxed, and the type of the return value is deduced by the compiler from the `return` expression. If there are multiple return values they should have the same type:

```
// C++11
auto func1(int const i) -> int
{ return 2*i; }

// C++14
auto func2(int const i)
{ return 2*i; }
```

As mentioned earlier, `auto` does not retain `const`/`volatile` and reference qualifiers. This leads to problems with `auto` as a placeholder for the return type from a function. To explain this, let us consider the preceding example with `foo.get()`. This time we have a wrapper function called `proxy_get()` that takes a reference to a `foo`, calls `get()`, and returns the value returned by `get()`, which is an `int&`. However, the compiler will deduce the return type of `proxy_get()` as being `int`, not `int&`. Trying to assign that value to an `int&` fails with an error:

```
class foo
{
  int x_;
public:
  foo(int const x = 0) :x_{ x } {}
  int& get() { return x_; }
};

auto proxy_get(foo& f) { return f.get(); }

auto f = foo{ 42 };
auto& x = proxy_get(f); // cannot convert from 'int' to 'int &'
```

To fix this, we need to actually return `auto&`. However, this is a problem with templates and perfect forwarding the return type without knowing whether that is a value or a reference. The solution to this problem in C++14 is `decltype(auto)` that will correctly deduce the type:

```
decltype(auto) proxy_get(foo& f) { return f.get(); }
auto f = foo{ 42 };
decltype(auto) x = proxy_get(f);
```

The last important case where `auto` can be used is with lambdas. As of C++14, both lambda return type and lambda parameter types can be `auto`. Such a lambda is called a *generic lambda* because the closure type defined by the lambda has a templated call operator. The following shows a generic lambda that takes two `auto` parameters and returns the result of applying `operator+` on the actual types:

```
auto ladd = [] (auto const a, auto const b) { return a + b; };
struct
{
   template<typename T, typename U>
   auto operator () (T const a, U const b) const { return a+b; }
} L;
```

This lambda can be used to add anything for which the `operator+` is defined. In the following example, we use the lambda to add two integers and to concatenate to `std::string` objects (using the C++14 user-defined literal `operator ""s`):

```
    auto i = ladd(40, 2);            // 42
    auto s = ladd("forty"s, "two"s); // "fortytwo"s
```

See also

- *Creating type aliases and alias templates*
- *Understanding uniform initialization*

Creating type aliases and alias templates

In C++, it is possible to create synonyms that can be used instead of a type name. This is achieved by creating a `typedef` declaration. This is useful in several cases, such as creating shorter or more meaningful names for a type or names for function pointers. However, `typedef` declarations cannot be used with templates to create `template type aliases`.
An `std::vector<T>`, for instance, is not a type (`std::vector<int>` is a type), but a sort of family of all types that can be created when the type placeholder `T` is replaced with an actual type.

In C++11, a type alias is a name for another already declared type, and an alias template is a name for another already declared template. Both of these types of aliases are introduced with a new `using` syntax.

How to do it…

- Create type aliases with the form `using identifier = type-id` as in the following examples:

```
using byte    = unsigned char;
using pbyte   = unsigned char *;
using array_t = int[10];
using fn      = void(byte, double);

void func(byte b, double d) { /*...*/ }

byte b {42};
pbyte pb = new byte[10] {0};
array_t a{0,1,2,3,4,5,6,7,8,9};
fn* f = func;
```

- Create alias templates with the form `template<template-params-list> identifier = type-id` as in the following examples:

```
template <class T>
class custom_allocator { /* ... */};

template <typename T>
using vec_t = std::vector<T, custom_allocator<T>>;

vec_t<int>           vi;
vec_t<std::string>   vs;
```

For consistency and readability, you should do the following:

- Not mix `typedef` and `using` declarations for creating aliases.
- Use the `using` syntax to create names of function pointer types.

How it works…

A `typedef` declaration introduces a synonym (or an alias in other words) for a type. It does not introduce another type (like a `class`, `struct`, `union`, or `enum` declaration). Type names introduced with a `typedef` declaration follow the same hiding rules as identifier names. They can also be redeclared, but only to refer to the same type (therefore, you can have valid multiple `typedef` declarations that introduce the same type name synonym in a translation unit as long as it is a synonym for the same type). The following are typical examples of `typedef` declarations:

```
typedef unsigned char   byte;
typedef unsigned char * pbyte;
typedef int             array_t[10];
typedef void(*fn)(byte, double);

template<typename T>
class foo {
  typedef T value_type;
};

typedef std::vector<int> vint_t;
```

A type alias declaration is equivalent to a `typedef` declaration. It can appear in a block scope, class scope, or namespace scope. According to C++11 paragraph 7.1.3.2:

A typedef-name can also be introduced by an alias-declaration. The identifier following the using keyword becomes a typedef-name and the optional attribute-specifier-seq following the identifier appertains to that typedef-name. It has the same semantics as if it were introduced by the typedef specifier. In particular, it does not define a new type and it shall not appear in the type-id.

An alias-declaration is, however, more readable and more clear about the actual type that is aliased when it comes to creating aliases for array types and function pointer types. In the examples from the *How to do it…* section, it is easily understandable that `array_t` is a name for the type array of 10 integers, and `fn` is a name for a function type that takes two parameters of type `byte` and `double` and returns `void`. That is also consistent with the syntax for declaring `std::function` objects (for example, `std::function<void(byte, double)> f`).

The driving purpose of the new syntax is to define alias templates. These are templates which, when specialized, are equivalent to the result of substituting the template arguments of the alias template for the template parameters in the `type-id`.

It is important to take note of the following things:

- Alias templates cannot be partially or explicitly specialized.
- Alias templates are never deduced by template argument deduction when deducing a template parameter.
- The type produced when specializing an alias template is not allowed to directly or indirectly make use of its own type.

Understanding uniform initialization

Brace-initialization is a uniform method for initializing data in C++11. For this reason, it is also called *uniform initialization*. It is arguably one of the most important features from C++11 that developers should understand and use. It removes previous distinctions between initializing fundamental types, aggregate and non-aggregate types, and arrays and standard containers.

Getting ready

For continuing with this recipe, you need to be familiar with direct initialization that initializes an object from an explicit set of constructor arguments and copy initialization that initializes an object from another object. The following is a simple example of both types of initialization, but for further details, you should see additional resources:

```
    std::string s1("test");   // direct initialization
    std::string s2 = "test";  // copy initialization
```

How to do it…

To uniformly initialize objects regardless of their type, use the brace-initialization form `{}` that can be used for both direct initialization and copy initialization. When used with brace initialization, these are called direct list and copy list initialization.

```
T object {other};   // direct list initialization
T object = {other}; // copy list initialization
```

Examples of uniform initialization are as follows:

- Standard containers:

```
std::vector<int> v { 1, 2, 3 };
std::map<int, std::string> m { {1, "one"}, { 2, "two" }};
```

- Dynamically allocated arrays:

```
int* arr2 = new int[3]{ 1, 2, 3 };
```

- Arrays:

```
int arr1[3] { 1, 2, 3 };
```

- Built-in types:

```
int i { 42 };
double d { 1.2 };
```

- User-defined types:

```
class foo
{
  int a_;
  double b_;
public:
  foo():a_(0), b_(0) {}
  foo(int a, double b = 0.0):a_(a), b_(b) {}
};

foo f1{};
foo f2{ 42, 1.2 };
foo f3{ 42 };
```

- User-defined POD types:

```
struct bar { int a_; double b_;};
bar b{ 42, 1.2 };
```

How it works…

Before C++11 objects required different types of initialization based on their type:

- Fundamental types could be initialized using assignment:

```
int a = 42;
double b = 1.2;
```

- Class objects could also be initialized using assignment from a single value if they had a conversion constructor (prior to C++11, a constructor with a single parameter was called a *conversion constructor*):

```
class foo
{
  int a_;
public:
  foo(int a):a_(a) {}
};
foo f1 = 42;
```

- Non-aggregate classes could be initialized with parentheses (the functional form) when arguments were provided and only without any parentheses when default initialization was performed (call to the default constructor). In the next example, `foo` is the structure defined in the *How to do it…* section:

```
foo f1;           // default initialization
foo f2(42, 1.2);
foo f3(42);
foo f4();         // function declaration
```

- Aggregate and POD types could be initialized with brace-initialization. In the next example, `bar` is the structure defined in the *How to do it…* section:

```
bar b = {42, 1.2};
int a[] = {1, 2, 3, 4, 5};
```

Apart from the different methods of initializing the data, there are also some limitations. For instance, the only way to initialize a standard container was to first declare an object and then insert elements into it; vector was an exception because it is possible to assign values from an array that can be prior initialized using aggregate initialization. On the other hand, however, dynamically allocated aggregates could not be initialized directly.

All the examples in the *How to do it…* section use direct initialization, but copy initialization is also possible with brace-initialization. The two forms, direct and copy initialization, may be equivalent in most cases, but copy initialization is less permissive because it does not consider explicit constructors in its implicit conversion sequence that must produce an object directly from the initializer, whereas direct initialization expects an implicit conversion from the initializer to an argument of the constructor. Dynamically allocated arrays can only be initialized using direct initialization.

Of the classes shown in the preceding examples, `foo` is the one class that has both a default

constructor and a constructor with parameters. To use the default constructor to perform default initialization, we need to use empty braces, that is, `{}`. To use the constructor with parameters, we need to provide the values for all the arguments in braces `{}`. Unlike non-aggregate types where default initialization means invoking the default constructor, for aggregate types, default initialization means initializing with zeros.

Initialization of standard containers, such as the vector and the map also shown above, is possible because all standard containers have an additional constructor in C++11 that takes an argument of type `std::initializer_list<T>`. This is basically a lightweight proxy over an array of elements of type `T const`. These constructors then initialize the internal data from the values in the initializer list.

The way the initialization using `std::initializer_list` works is the following:

- The compiler resolves the types of the elements in the initialization list (all elements must have the same type).
- The compiler creates an array with the elements in the initializer list.
- The compiler creates an `std::initializer_list<T>` object to wrap the previously created array.
- The `std::initializer_list<T>` object is passed as an argument to the constructor.

An initializer list always takes precedence over other constructors where brace-initialization is used. If such a constructor exists for a class, it will be called when brace-initialization is performed:

```
class foo
{
  int a_;
  int b_;
public:
  foo() :a_(0), b_(0) {}

  foo(int a, int b = 0) :a_(a), b_(b) {}
  foo(std::initializer_list<int> l) {}
};

foo f{ 1, 2 }; // calls constructor with initializer_list<int>
```

The precedence rule applies to any function, not just constructors. In the following example, two overloads of the same function exist. Calling the function with an initializer list resolves to a call to the overload with an `std::initializer_list`:

```
void func(int const a, int const b, int const c)
{
  std::cout << a << b << c << std::endl;
}

void func(std::initializer_list<int> const l)
{
  for (auto const & e : l)
    std::cout << e << std::endl;
}

func({ 1,2,3 }); // calls second overload
```

This, however, has the potential of leading to bugs. Let’s take, for example, the vector type. Among the constructors of the vector, there is one that has a single argument representing the initial number of elements to be allocated and another one that has

an `std::initializer_list` as an argument. If the intention is to create a vector with a preallocated size, using the brace-initialization will not work, as the constructor with the `std::initializer_list` will be the best overload to be called:

```
    std::vector<int> v {5};
```

The preceding code does not create a vector with five elements, but a vector with one element with a value 5. To be able to actually create a vector with five elements, initialization with the parentheses form must be used:

```
    std::vector<int> v (5);
```

Another thing to note is that brace-initialization does not allow narrowing conversion. According to the C++ standard (refer to paragraph 8.5.4 of the standard), a narrowing conversion is an implicit conversion:

- From a floating-point type to an integer type
- From long double to double or float, or from double to float, except where the source is a constant expression and the actual value after conversion is within the range of values that can be represented (even if it cannot be represented exactly)
- From an integer type or unscoped enumeration type to a floating-point type, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted to its original type
- From an integer type or unscoped enumeration type to an integer type that cannot represent all the values of the original type, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted to its original type.

The following declarations trigger compiler errors because they require a narrowing conversion:

```
    int i{ 1.2 };           // error

    double d = 47 / 13;
    float f1{ d };          // error
    float f2{47/13};        // OK
```

To fix the error, an explicit conversion must be done:

```
    int i{ static_cast<int>(1.2) };

    double d = 47 / 13;
    float f1{ static_cast<float>(d) };
```

A brace-initialization list is not an expression and does not have a type. Therefore, `decltype` cannot be used on a brace-init list, and template type deduction cannot deduce the type that matches a brace-init list.

There’s more

The following sample shows several examples of direct-list-initialization and copy-list-initialization. In C++11, the deduced type of all these expressions is `std::initializer_list<int>`.

```
auto a = {42};   // std::initializer_list<int>
auto b {42};     // std::initializer_list<int>
auto c = {4, 2}; // std::initializer_list<int>
auto d {4, 2};   // std::initializer_list<int>
```

C++17 has changed the rules for list initialization, differentiating between the direct- and copy-list-initialization. The new rules for type deduction are as follows:

- for copy list initialization auto deduction will deduce a `std::initializer_list<T>` if all elements in the list have the same type, or be ill-formed.
- for direct list initialization auto deduction will deduce a `T` if the list has a single element, or be ill-formed if there is more than one element.

Base on the new rules, the previous examples would change as follows: `a` and `c` are deduced as `std::initializer_list<int>`; `b` is deduced as an `int`; `d`, which uses direct initialization and has more than one value in the brace-init-list, triggers a compiler error.

```
auto a = {42};   // std::initializer_list<int>
auto b {42};     // int
auto c = {4, 2}; // std::initializer_list<int>
auto d {4, 2};   // error, too many
```

See also

- *Using auto whenever possible*
- *Understanding the various forms of non-static member initialization*

Understanding the various forms of non-static member initialization

Constructors are a place where non-static class member initialization is done. Many developers prefer assignments in the constructor body. Aside from the several exceptional cases when that is actually necessary, initialization of non-static members should be done in the constructor's initializer list or, as of C++11, using default member initialization when they are declared in the class. Prior to C++11, constants and non-constant non-static data members of a class had to be initialized in the constructor. Initialization on declaration in a class was only possible for static constants. As we will see further, this limitation was removed in C++11 that allows initialization of non-statics in the class declaration. This initialization is called *default member initialization* and is explained in the next sections.

This recipe will explore the ways the non-static member initialization should be done.

How to do it…

To initialize non-static members of a class you should:

- Use default member initialization for providing default values for members of classes with multiple constructors that would use a common initializer for those members (see `[3]` and `[4]` in the following code).
- Use default member initialization for constants, both static and non-static (see `[1]` and `[2]` in the following code).
- Use the constructor initializer list to initialize members that don't have default values, but depend on constructor parameters (see `[5]` and `[6]` in the following code).
- Use assignment in constructors when the other options are not possible (examples include initializing data members with pointer `this`, checking constructor parameter values, and throwing exceptions prior to initializing members with those values or self-references of two non-static data members).

The following example shows these forms of initialization:

```
struct Control
{
  const int DefaultHeigh = 14;                  // [1]
  const int DefaultWidth = 80;                  // [2]

  TextVAligment valign = TextVAligment::Middle; // [3]
  TextHAligment halign = TextHAligment::Left;   // [4]

  std::string text;

  Control(std::string const & t) : text(t)       // [5]
  {}

  Control(std::string const & t,
    TextVerticalAligment const va,
    TextHorizontalAligment const ha):
  text(t), valign(va), halign(ha)                // [6]
  {}
};
```

How it works…

Non-static data members are supposed to be initialized in the constructor's initializer list as shown in the following example:

```
struct Point
{
  double X, Y;
  Point(double const x = 0.0, double const y = 0.0) : X(x), Y(y)  {}
};
```

Many developers, however, do not use the initializer list, but prefer assignments in the constructor's body, or even mix assignments and the initializer list. That could be for several reasons—for larger classes with many members, the constructor assignments may look easier to read than long initializer lists, perhaps split on many lines, or it could be because they are familiar with other programming languages that don't have an initializer list or because, unfortunately, for various reasons they don't even know about it.

It is important to note that the order in which non-static data members are initialized is the order in which they were declared in the class definition, and not the order of their initialization in a constructor initializer list. On the other hand, the order in which non-static data members are destroyed is the reversed order of construction.

Using assignments in the constructor is not efficient, as this can create temporary objects that are later discarded. If not initialized in the initializer list, non-static members are initialized via their default constructor and then, when assigned a value in the constructor's body, the assignment operator is invoked. This can lead to inefficient work if the default constructor allocates a resource (such as memory or a file) and that has to be deallocated and reallocated in the assignment operator:

```
struct foo
{
  foo()
  { std::cout << "default constructor" << std::endl; }
  foo(std::string const & text)
  { std::cout << "constructor '" << text << "'" << std::endl; }
  foo(foo const & other)
  { std::cout << "copy constructor" << std::endl; }
  foo(foo&& other)
  { std::cout << "move constructor" << std::endl; };
  foo& operator=(foo const & other)
  { std::cout << "assignment" << std::endl; return *this; }
  foo& operator=(foo&& other)
  { std::cout << "move assignment" << std::endl; return *this;}
  ~foo()
  { std::cout << "destructor" << std::endl; }
};

struct bar
{
  foo f;

  bar(foo const & value)
  {
    f = value;
  }
};

foo f;
```

```
    bar b(f);
```

The preceding code produces the following output showing how data member `f` is first default initialized and then assigned a new value:

```
default constructor
default constructor
assignment
destructor
destructor
```

Changing the initialization from the assignment in the constructor body to the initializer list replaces the calls to the default constructor plus assignment operator with a call to the copy constructor:

```
    bar(foo const & value) : f(value) { }
```

Adding the preceding line of code produces the following output:

```
default constructor
copy constructor
destructor
destructor
```

For those reasons, at least for other types than the built-in types (such as `bool`, `char`, `int`, `float`, `double` or pointers), you should prefer the constructor initializer list. However, to be consistent with your initialization style, you should always prefer the constructor initializer list when that is possible. There are several situations when using the initializer list is not possible; these include the following cases (but the list could be expanded with other cases):

- If a member has to be initialized with a pointer or reference to the object that contains it, using the `this` pointer in the initialization list may trigger a warning with some compilers that it is used before the object is constructed.
- If you have two data members that must contain references to each other.
- If you want to test an input parameter and throw an exception before initializing a non-static data member with the value of the parameter.

Starting with C++11, non-static data members can be initialized when declared in the class. This is called *default member initialization* because it is supposed to represent initialization with default values. Default member initialization is intended for constants and for members that are not initialized based on constructor parameters (in other words members whose value does not depend on the way the object is constructed):

```
    enum class TextFlow { LeftToRight, RightToLeft };

    struct Control
    {
      const int DefaultHeight = 20;
      const int DefaultWidth = 100;

      TextFlow textFlow = TextFlow::LeftToRight;
      std::string text;

      Control(std::string t) : text(t)
      {}
    };
```

In the preceding example, `DefaultHeight` and `DefaultWidth` are both constants; therefore, the

values do not depend on the way the object is constructed, so they are initialized when declared. The `textFlow` object is a non-constant non-static data member whose value also does not depend on the way the object is initialized (it could be changed via another member function), therefore, it is also initialized using default member initialization when it is declared. `text`, on the other hand, is also a non-constant non-static data member, but its initial value depends on the way the object is constructed and therefore it is initialized in the constructor's initializer list using a value passed as an argument to the constructor.

If a data member is initialized both with the default member initialization and constructor initializer list, the latter takes precedence and the default value is discarded. To exemplify this, let's again consider the `foo` class earlier and the following `bar` class that uses it:

```
struct bar
{
  foo f{"default value"};

  bar() : f{"constructor initializer"}
  {
  }
};

bar b;
```

The output differs, in this case, as follows, because the value from the default initializer list is discarded, and the object is not initialized twice:

```
constructor
constructor initializer
destructor
```

Using the appropriate initialization method for each member leads not only to more efficient code but also to better organized and more readable code.

Controlling and querying object alignment

C++11 provides standardized methods for specifying and querying the alignment requirements of a type (something that was previously possible only through compiler-specific methods). Controlling the alignment is important in order to boost performance on different processors and enable the use of some instructions that only work with data on particular alignments. For example, Intel SSE and Intel SSE2 require 16 bytes alignment of data, whereas for Intel Advanced Vector Extensions (or Intel AVX), it is highly recommended to use 32 bytes alignment. This recipe explores the `alignas` specifier for controlling the alignment requirements and the `alignof` operator that retrieves the alignment requirements of a type.

Getting ready

You should be familiar with what data alignment is and the way the compiler performs default data alignment. However, basic information about the latter is provided in the *How it works…* section.

How to do it…

- To control the alignment of a type (both at the class level or data member level) or an object, use the `alignas` specifier:

```
struct alignas(4) foo
{
  char a;
  char b;
};
struct bar
{
  alignas(2) char a;
  alignas(8) int  b;
};
alignas(8)   int a;
alignas(256) long b[4];
```

- To query the alignment of a type, use the `alignof` operator:

```
auto align = alignof(foo);
```

How it works…

Processors do not access memory one byte at a time, but in larger chunks of powers of twos (2, 4, 8, 16, 32, and so on). Owing to this, it is important that compilers align data in memory so that it can be easily accessed by the processor. Should this data be misaligned, the compiler has to do extra work for accessing data; it has to read multiple chunks of data, shift, and discard unnecessary bytes and combine the rest together.

C++ compilers align variables based on the size of their data type: 1 byte for `bool` and `char`, 2 bytes for `short`, 4 bytes for `int`, `long` and `float`, 8 bytes for `double` and `long long`, and so on. When it comes to structures or unions, the alignment must match the size of the largest member in order to avoid performance issues. To exemplify, let's consider the following data structures:

```
struct foo1    // size = 1, alignment = 1
{
  char a;
};

struct foo2    // size = 2, alignment = 1
{
  char a;
  char b;
};

struct foo3    // size = 8, alignment = 4
{
  char a;
  int  b;
};
```

`foo1` and `foo2` have different sizes, but the alignment is the same—that is, 1—because all data members are of type `char`, which has a size of 1. In structure `foo3`, the second member is an integer, whose size is 4. As a result, the alignment of members of this structure is done at addresses that are multiples of 4. To achieve that, the compiler introduces padding bytes. The structure `foo3` is actually transformed into the following:

```
struct foo3_
{
  char a;        // 1 byte
  char _pad0[3]; // 3 bytes padding to put b on a 4-byte boundary
  int  b;        // 4 bytes
};
```

Similarly, the following structure has a size of 32 bytes and an alignment of 8; that is because the largest member is a `double` whose size is 8. This structure, however, requires padding in several places to make sure that all members can be accessed at addresses that are multiples of 8:

```
struct foo4
{
  int a;
  char b;
  float c;
  double d;
  bool e;
};
```

The equivalent structure created by the compiler is as follows:

```
struct foo4_
{
  int a;         // 4 bytes
  char b;        // 1 byte
  char _pad0[3]; // 3 bytes padding to put c on a 8-byte boundary
  float c;       // 4 bytes
  char _pad1[4]; // 4 bytes padding to put d on a 8-byte boundary
  double d;      // 8 bytes
  bool e;        // 1 byte
  char _pad2[7]; // 7 bytes padding to make sizeof struct multiple of 8
};
```

In C++11, specifying the alignment of an object or type is done using the `alignas` specifier. This can take either an expression (an integral constant expression that evaluates to 0 or a valid value for an alignment), a type-id, or a parameter pack. The `alignas` specifier can be applied to the declaration of a variable or a class data member that does not represent a bit field, or to the declaration of a class, union, or enumeration. The type or object on which an `alignas` specification is applied will have the alignment requirement equal to the largest, greater than zero, expression of all `alignas` specifications used in the declaration.

There are several restrictions when using the `alignas` specifier:

- The only valid alignments are the powers of two (1, 2, 4, 8, 16, 32, and so on). Any other values are illegal, and the program is considered ill-formed; that doesn't necessarily have to produce an error, as the compiler may choose to ignore the specification.
- An alignment of 0 is always ignored.
- If the largest `alignas` on a declaration is smaller than the natural alignment without any `alignas` specifier, then the program is also considered ill-formed.

In the following example, the `alignas` specifier is applied on a class declaration. The natural alignment without the `alignas` specifier would have been 1, but with `alignas(4)` it becomes 4:

```
struct alignas(4) foo
{
  char a;
  char b;
};
```

In other words, the compiler transforms the preceding class into the following:

```
struct foo
{
  char a;
  char b;
  char _pad0[2];
};
```

The `alignas` specifier can be applied both on the class declaration and the member data declarations. In this case, the strictest (that is, largest) value wins. In the following example, member `a` has a natural size of 1 and requires an alignment of 2; member `b` has a natural size of 4 and requires an alignment of 8, therefore, the strictest alignment would be 8. The alignment requirement of the entire class is 4, which is weaker (that is, smaller) than the strictest required alignment and therefore it will be ignored, though the compiler will produce a warning:

```
struct alignas(4) foo
{
  alignas(2) char a;
```

```
  alignas(8) int  b;
};
```

The result is a structure that looks like this:

```
struct foo
{
  char a;
  char _pad0[7];
  int b;
  char _pad1[4];
};
```

The `alignas` specifier can also be applied on variables. In the next example, variable `a`, that is an integer, is required to be placed in memory at a multiple of 8. The next variable, the array of 4 `a`, that is an integer, is required to be placed in memory at a multiple of 8. The next variable, the array of 4 `long`s, is required to be placed in memory at a multiple of 256. As a result, the compiler will introduce up to 244 bytes of padding between the two variables (depending on where in memory, at an address multiple of 8, the variable `a` is located):

```
alignas(8)   int a;
alignas(256) long b[4];

printf("%pn", &a); // eg. 0000006C0D9EF908
printf("%pn", &b); // eg. 0000006C0D9EFA00
```

Looking at the addresses, we can see that the address of `a` is indeed a multiple of 8, and the address of `b` is a multiple of 256 (hexadecimal 100).

To query the alignment of a type, we use the `alignof` operator. Unlike `sizeof`, this operator can only be applied to type-ids, and not on variables or class data members. The types on which it can be applied can be complete types, an array type, or a reference type. For arrays, the value returned is the alignment of the element type; for references, the value returned is the alignment of the referenced type. Here are several examples:

Expression	**Evaluation**
`alignof(char)`	1, because the natural alignment of `char` is 1
`alignof(int)`	4, because the natural alignment of `int` is 4
`alignof(int*)`	4 on 32-bit, 8 on 64-bit, the alignment for pointers
`alignof(int[4])`	4, because the natural alignment of the element type is 4
`alignof(foo&)`	8, because the specified alignment for class `foo` that is the referred type (as shown in the last example) was 8

Using scoped enumerations

Enumeration is a basic type in C++ that defines a collection of values, always of an integral underlying type. Their named values, that are constant, are called enumerators. Enumerations declared with keyword `enum` are called *unscoped enumerations* and enumerations declared with `enum class` or `enum struct` are called *scoped enumerations*. The latter ones were introduced in C++11 and are intended to solve several problems of the unscoped enumerations.

How to do it…

- Prefer to use scoped enumerations instead of unscoped ones.
- In order to use scoped enumerations, you should declare enumerations using `enum class` or `enum struct`:

```
enum class Status { Unknown, Created, Connected };
Status s = Status::Created;
```

The `enum class` and `enum struct` declarations are equivalent, and throughout this recipe and the rest of the book, we will use `enum class`.

How it works…

Unscoped enumerations have several issues that are creating problems for developers:

- They export their enumerators to the surrounding scope (for which reason, they are called unscoped enumerations), and that has the following two drawbacks: it can lead to name clashes if two enumerations in the same namespace have enumerators with the same name, and it's not possible to use an enumerator using its fully qualified name:

```
enum Status {Unknown, Created, Connected};
enum Codes {OK, Failure, Unknown};   // error
auto status = Status::Created;       // error
```

- Prior to C++ 11, they could not specify the underlying type that is required to be an integral type. This type must not be larger than `int`, unless the enumerator value cannot fit a signed or unsigned integer. Owing to this, forward declaration of enumerations was not possible. The reason was that the size of the enumeration was not known since the underlying type was not known until values of the enumerators were defined so that the compiler could pick the appropriate integer type. This has been fixed in C++11.
- Values of enumerators implicitly convert to `int`. That means you can intentionally or accidentally mix enumerations that have a certain meaning and integers (that may not even be related to the meaning of the enumeration) and the compiler will not be able to warn you:

```
enum Codes { OK, Failure };
void include_offset(int pixels) {/*...*/}
include_offset(Failure);
```

The scoped enumerations are basically strongly typed enumerations that behave differently than the unscoped enumerations:

- They do not export their enumerators to the surrounding scope. The two enumerations shown earlier would change to the following, no longer generating a name collision and being possible to fully qualify the names of the enumerators:

```
enum class Status { Unknown, Created, Connected };
enum class Codes { OK, Failure, Unknown }; // OK
Codes code = Codes::Unknown;               // OK
```

- You can specify the underlying type. The same rules for underlying types of unscoped enumerations apply to scoped enumerations too, except that the user can specify explicitly the underlying type. This also solves the problem with forward declarations since the underlying type can be known before the definition is available:

```
enum class Codes : unsigned int;

void print_code(Codes const code) {}
```

```
enum class Codes : unsigned int
{
   OK = 0,
   Failure = 1,
   Unknown = 0xFFFF0000U
};
```

- Values of scoped enumerations no longer convert implicitly to `int`. Assigning the value of an `enum class` to an integer variable would trigger a compiler error unless an explicit cast is specified:

```
Codes c1 = Codes::OK;                       // OK
int c2 = Codes::Failure;                    // error
int c3 = static_cast<int>(Codes::Failure);  // OK
```

Using override and final for virtual methods

Unlike other similar programming languages, C++ does not have a specific syntax for declaring interfaces (that are basically classes with pure virtual methods only) and also has some deficiencies related to how virtual methods are declared. In C++, the virtual methods are introduced with the `virtual` keyword. However, the keyword `virtual` is optional for declaring overrides in derived classes that can lead to confusion when dealing with large classes or hierarchies. You may need to navigate throughout the hierarchy up to the base to figure out whether a function is virtual or not. On the other hand, sometimes, it is useful to make sure that a virtual function or even a derived class can no longer be overridden or derived further. In this recipe, we will see how to use C++11 special identifiers `override` and `final` to declare virtual functions or classes.

Getting ready

You should be familiar with inheritance and polymorphism in C++ and concepts, such as abstract classes, pure specifiers, virtual, and overridden methods.

How to do it…

To ensure correct declaration of virtual methods both in base and derived classes, but also increase readability, do the following:

- Always use the `virtual` keyword when declaring virtual functions in derived classes that are supposed to override virtual functions from a base class, and
- Always use the `override` special identifier after the declarator part of a virtual function declaration or definition.

```
class Base
{
  virtual void foo() = 0;
  virtual void bar() {}
  virtual void foobar() = 0;
};

void Base::foobar() {}

class Derived1 : public Base
{
  virtual void foo() override = 0;
  virtual void bar() override {}
  virtual void foobar() override {}
};

class Derived2 : public Derived1
{
  virtual void foo() override {}
};
```

The declarator is the part of the type of a function that excludes the return type.

To ensure that functions cannot be overridden further or classes cannot be derived any more, use the `final` special identifier:

- After the declarator part of a virtual function declaration or definition to prevent further overrides in a derived class:

```
class Derived2 : public Derived1
{
  virtual void foo() final {}
};
```

- After the name of a class in the declaration of the class to prevent further derivations of the class:

```
class Derived4 final : public Derived1
{
  virtual void foo() override {}
};
```

How it works…

The way `override` works is very simple; in a virtual function declaration or definition, it ensures that the function is actually overriding a base class function, otherwise, the compiler will trigger an error.

It should be noted that both `override` and `final` keywords are special identifiers having a meaning only in a member function declaration or definition. They are not reserved keywords and can still be used elsewhere in a program as user-defined identifiers.

Using the `override` special identifier helps the compiler to detect situations when a virtual method does not override another one like shown in the following example:

```
class Base
{
public:
  virtual void foo() {}
  virtual void bar() {}
};

class Derived1 : public Base
{
public:
  void foo() override {}
  // for readability use the virtual keyword

  virtual void bar(char const c) override {}
  // error, no Base::bar(char const)
};
```

The other special identifier, `final`, is used in a member function declaration or definition to indicate that the function is virtual and cannot be overridden in a derived class. If a derived class attempts to override the virtual function, the compiler triggers an error:

```
class Derived2 : public Derived1
{
  virtual void foo() final {}
};

class Derived3 : public Derived2
{
  virtual void foo() override {} // error
};
```

The `final` specifier can also be used in a class declaration to indicate that it cannot be derived:

```
class Derived4 final : public Derived1
{
  virtual void foo() override {}
};

class Derived5 : public Derived4 // error
{
};
```

Since both `override` and `final` have this special meaning when used in the defined context and are not in fact reserved keywords, you can still use them anywhere elsewhere in the C++ code. This ensured that existing code written before C++11 did not break because of the use of these names for identifiers:

```
class foo
```

```
    {
      int final = 0;
      void override() {}
    };
```

Using range-based for loops to iterate on a range

Many programming languages support a variant of a `for` loop called `for each`, that is, repeating a group of statements over the elements of a collection. C++ did not have core language support for this until C++11. The closest feature was the general purpose algorithm from the standard library called `std::for_each`, that applies a function to all the elements in a range. C++11 brought language support for `for each` that is actually called *range-based for loops*. The new C++17 standard provides several improvements to the original language feature.

Getting ready

In C++11, a range-based for loop has the following general syntax:

```
for ( range_declaration : range_expression ) loop_statement
```

To exemplify the various ways of using a range-based for loops, we will use the following functions that return sequences of elements:

```
std::vector<int> getRates()
{
  return std::vector<int> {1, 1, 2, 3, 5, 8, 13};
}

std::multimap<int, bool> getRates2()
{
  return std::multimap<int, bool> {
    { 1, true },
    { 1, true },
    { 2, false },
    { 3, true },
    { 5, true },
    { 8, false },
    { 13, true }
  };
}
```

How to do it…

Range-based for loops can be used in various ways:

- By committing to a specific type for the elements of the sequence:

```
auto rates = getRates();
for (int rate : rates)
  std::cout << rate << std::endl;
for (int& rate : rates)
  rate *= 2;
```

- By not specifying a type and letting the compiler deduce it:

```
for (auto&& rate : getRates())
  std::cout << rate << std::endl;

for (auto & rate : rates)
  rate *= 2;

for (auto const & rate : rates)
  std::cout << rate << std::endl;
```

- By using structured bindings and decomposition declaration in C++17:

```
for (auto&& [rate, flag] : getRates2())
  std::cout << rate << std::endl;
```

How it works…

The expression for the range-based for loops shown earlier in the *How to do it…* section is basically syntactic sugar as the compiler transforms it into something else. Before C++17, the code generated by the compiler used to be the following:

```
{
  auto && __range = range_expression;
  for (auto __begin = begin_expr, __end = end_expr;
  __begin != __end; ++__begin) {
    range_declaration = *__begin;
    loop_statement
  }
}
```

What `begin_expr` and `end_expr` are in this code depends on the type of the range:

- For C-like arrays, `__range` and `__bound` are the number of elements in the array.
- For a class type with `begin()` and `end()` members (regardless of their type and accessibility): `__range.begin()` and `__range.end()`.
- For others it is `begin(__range)` and `end(__range)` that are determined via argument dependent lookup.

It is important to note that if a class contains any members (function, data member, or enumerators) called `begin` or `end`, regardless of their type and accessibility, they will be picked for `begin_expr` and `end_expr`. Therefore, such a class type cannot be used in range-based for loops.

In C++17, the code generated by the compiler is slightly different:

```
{
  auto && __range = range_expression;
  auto __begin = begin_expr;
  auto __end = end_expr;
  for (; __begin != __end; ++__begin) {
    range_declaration = *__begin;
    loop_statement
  }
}
```

The new standard has removed the constraint that the begin expression and end expression must have the same type. The end expression does not need to be an actual iterator, but it has to be able to be compared for inequality with an iterator. A benefit of this is that the range can be delimited by a predicate.

See also

- *Enabling range-based for loops for custom types*

Enabling range-based for loops for custom types

As we have seen in the preceding recipe, the range-based for loops, known as `for each` in other programming languages, allows you to iterate over the elements of a range, providing a simplified syntax over the standard `for` loops and making the code more readable in many situations. However, range-based for loops do not work out of the box with any type representing a range, but require the presence of a `begin()` and `end()` function (for non-array types) either as a member or free function. In this recipe, we will see how to enable a custom type to be used in range-based for loops.

Getting ready

It is recommended that you read the recipe *Using range-based for loops to iterate on a range* before continuing with this one if you need to understand how range-based for loops work and what is the code the compiler generates for such a loop.

To show how we can enable range-based for loops for custom types representing sequences, we will use the following implementation of a simple array:

```
template <typename T, size_t const Size>
class dummy_array
{
  T data[Size] = {};

public:
  T const & GetAt(size_t const index) const
  {
    if (index < Size) return data[index];
    throw std::out_of_range("index out of range");
  }

  void SetAt(size_t const index, T const & value)
  {
    if (index < Size) data[index] = value;
    else throw std::out_of_range("index out of range");
  }

  size_t GetSize() const { return Size; }
};
```

The purpose of this recipe is to enable writing code like the following:

```
dummy_array<int, 3> arr;
arr.SetAt(0, 1);
arr.SetAt(1, 2);
arr.SetAt(2, 3);

for(auto&& e : arr)
{
  std::cout << e << std::endl;
}
```

How to do it…

To enable a custom type to be used in range-based `for` loops, you need to do the following:

- Create mutable and constant iterators for the type that must implement the following operators:
  - `operator++` for incrementing the iterator.
  - `operator*` for dereferencing the iterator and accessing the actual element pointed by the iterator.
  - `operator!=` for comparing with another iterator for inequality.
- Provide free `begin()` and `end()` functions for the type.

Given the earlier example of a simple range, we need to provide the following:

1. The following minimal implementation of an iterator class:

```
template <typename T, typename C, size_t const Size>
class dummy_array_iterator_type
{
public:
  dummy_array_iterator_type(C& collection,
                            size_t const index) :
  index(index), collection(collection)
  { }

bool operator!= (dummy_array_iterator_type const & other) const
{
  return index != other.index;
}

T const & operator* () const
{
  return collection.GetAt(index);
}

dummy_array_iterator_type const & operator++ ()
{
  ++index;
  return *this;
}

private:
  size_t   index;
  C&       collection;
};
```

2. Alias templates for mutable and constant iterators:

```
template <typename T, size_t const Size>
using dummy_array_iterator =
   dummy_array_iterator_type<
     T, dummy_array<T, Size>, Size>;

template <typename T, size_t const Size>
using dummy_array_const_iterator =
   dummy_array_iterator_type<
     T, dummy_array<T, Size> const, Size>;
```

3. Free `begin()` and `end()` functions that return the corresponding begin and end iterators,

with overloads for both alias templates:

```
template <typename T, size_t const Size>
inline dummy_array_iterator<T, Size> begin(
  dummy_array<T, Size>& collection)
{
  return dummy_array_iterator<T, Size>(collection, 0);
}

template <typename T, size_t const Size>
inline dummy_array_iterator<T, Size> end(
  dummy_array<T, Size>& collection)
{
  return dummy_array_iterator<T, Size>(
    collection, collection.GetSize());
}

template <typename T, size_t const Size>
inline dummy_array_const_iterator<T, Size> begin(
  dummy_array<T, Size> const & collection)
{
  return dummy_array_const_iterator<T, Size>(
    collection, 0);
}

template <typename T, size_t const Size>
inline dummy_array_const_iterator<T, Size> end(
  dummy_array<T, Size> const & collection)
{
  return dummy_array_const_iterator<T, Size>(
    collection, collection.GetSize());
}
```

How it works…

Having this implementation available, the range-based for loop shown earlier compiles and executes as expected. When performing argument dependent lookup, the compiler will identify the two `begin()` and `end()` functions that we wrote (that take a reference to a `dummy_array`) and therefore the code it generates becomes valid.

In the preceding example, we have defined one iterator class template and two alias templates, called `dummy_array_iterator` and `dummy_array_const_iterator`. The `begin()` and `end()` functions both have two overloads for these two types of iterators. This is necessary so that the container we have considered could be used in range-based for loops with both constant and non-constant instances:

```
template <typename T, const size_t Size>
void print_dummy_array(dummy_array<T, Size> const & arr)
{
  for (auto && e : arr)
  {
    std::cout << e << std::endl;
  }
}
```

A possible alternative to enable range-based for loops for the simple range class we considered for this recipe is to provide member `begin()` and `end()` functions. In general, that could make sense only if you own and can modify the source code. On the other hand, the solution shown in this recipe works in all cases and should be preferred to other alternatives.

See also

- *Creating type aliases and alias templates*

Using explicit constructors and conversion operators to avoid implicit conversion

Before C++11, a constructor with a single parameter was considered a converting constructor. With C++11, every constructor without the `explicit` specifier is considered a converting constructor. Such a constructor defines an implicit conversion from the type or types of its arguments to the type of the class. Classes can also define converting operators that convert the type of the class to another specified type. All these are useful in some cases, but can create problems in other cases. In this recipe, we will see how to use explicit constructors and conversion operators.

Getting ready

For this recipe, you need to be familiar with converting constructors and converting operators. In this recipe, you will learn how to write explicit constructors and conversion operators to avoid implicit conversions to and from a type. The use of explicit constructors and conversion operators (called *user-defined conversion functions*) enables the compiler to yield errors—that in some cases are coding errors—and allow developers to spot those errors quickly and fix them.

How to do it…

To declare explicit constructors and conversion operators (regardless of whether they are functions or function templates), use the `explicit` specifier in the declaration.

The following example shows both an explicit constructor and a converting operator:

```
struct handle_t
{
  explicit handle_t(int const h) : handle(h) {}

  explicit operator bool() const { return handle != 0; };
private:
  int handle;
};
```

How it works…

To understand why explicit constructors are necessary and how they work, we will first look at converting constructors. The following class has three constructors: a default constructor (without parameters), a constructor that takes an `int`, and a constructor that takes two parameters, an `int` and a `double`. They don't do anything, except printing a message. As of C++11, these are all considered converting constructors. The class also has a conversion operator that converts the type to a `bool`:

```
struct foo
{
  foo()
  { std::cout << "foo" << std::endl; }
  foo(int const a)
  { std::cout << "foo(a)" << std::endl; }
  foo(int const a, double const b)
  { std::cout << "foo(a, b)" << std::endl; }

  operator bool() const { return true; }
};
```

Based on this, the following definitions of objects are possible (note that the comments represent the console output):

```
foo f1;              // foo
foo f2 {};           // foo

foo f3(1);           // foo(a)
foo f4 = 1;          // foo(a)
foo f5 { 1 };        // foo(a)
foo f6 = { 1 };      // foo(a)

foo f7(1, 2.0);      // foo(a, b)
foo f8 { 1, 2.0 };   // foo(a, b)
foo f9 = { 1, 2.0 }; // foo(a, b)
```

`f1` and `f2` invoke the default constructor. `f3`, `f4`, `f5`, and `f6` invoke the constructor that takes an `int`. Note that all the definitions of these objects are equivalent, even if they look different (`f3` is initialized using the functional form, `f4` and `f6` are copy initialized, and `f5` is directly initialized using brace-init-list). Similarly, `f7`, `f8`, and `f9` invoke the constructor with two parameters.

It may be important to note that if `foo` defines a constructor that takes an `std::initializer_list`, then all the initializations using `{}` would resolve to that constructor:

```
foo(std::initializer_list<int> l)
{ std::cout << "foo(l)" << std::endl; }
```

In this case, `f5` and `f6` will `print foo(l)`, while `f8` and `f9` will generate compiler errors because all elements of the initializer list should be integers.

These may all look right, but the implicit conversion constructors enable scenarios where the implicit conversion may not be what we wanted:

```
void bar(foo const f)
{
}

bar({});             // foo()
bar(1);              // foo(a)
bar({ 1, 2.0 });     // foo(a, b)
```

The conversion operator to `bool` in the example above also enables us to use `foo` objects where boolean values are expected:

```
bool flag = f1;
if(f2) {}
std::cout << f3 + f4 << std::endl;
if(f5 == f6) {}
```

The first two are examples where `foo` is expected to be used as boolean but the last two with addition and test for equality are probably incorrect, as we most likely expect to add `foo` objects and test `foo` objects for equality, not the booleans they implicitly convert to.

Perhaps a more realistic example to understand where problems could arise would be to consider a string buffer implementation. This would be a class that contains an internal buffer of characters. The class may provide several conversion constructors: a default constructor, a constructor that takes a `size_t` parameter representing the size of the buffer to preallocate, and a constructor that takes a pointer to `char` that should be used to allocate and initialize the internal buffer. Succinctly, such a string buffer could look like this:

```
class string_buffer
{
public:
  string_buffer() {}

  string_buffer(size_t const size) {}

  string_buffer(char const * const ptr) {}

  size_t size() const { return ...; }
  operator bool() const { return ...; }
  operator char * const () const { return ...; }
};
```

Based on this definition, we could construct the following objects:

```
std::shared_ptr<char> str;
string_buffer sb1;             // empty buffer
string_buffer sb2(20);         // buffer of 20 characters
string_buffer sb3(str.get());
// buffer initialized from input parameter
```

`sb1` is created using the default constructor and thus has an empty buffer; `sb2` is initialized using the constructor with a single parameter and the value of the parameter represents the size in characters of the internal buffer; `sb3` is initialized with an existing buffer and that is used to define the size of the internal buffer and to copy its value into the internal buffer. However, the same definition also enables the following object definitions:

```
enum ItemSizes {DefaultHeight, Large, MaxSize};

string_buffer b4 = 'a';
string_buffer b5 = MaxSize;
```

In this case, `b4` is initialized with a `char`. Since an implicit conversion to `size_t` exists, the constructor with a single parameter will be called. The intention here is not necessarily clear; perhaps it should have been `"a"` instead of `'a'`, in which case the third constructor would have been called. However, `b5` is most likely an error, because `MaxSize` is an enumerator representing an `ItemSizes` and should have nothing to do with a string buffer size. These erroneous situations are not flagged by the compiler in any way.

Using the `explicit` specifier in the declaration of a constructor, that constructor becomes an

explicit constructor and no longer allows implicit constructions of objects of a class type. To exemplify this, we will slightly change the `string_buffer` class earlier to declare all constructors explicit:

```
class string_buffer
{
public:
  explicit string_buffer() {}

  explicit string_buffer(size_t const size) {}

  explicit string_buffer(char const * const ptr) {}

  explicit operator bool() const { return ...; }
  explicit operator char * const () const { return ...; }
};
```

The change is minimal, but the definitions of `b4` and `b5` in the earlier example no longer work, and are incorrect, since the implicit conversion from `char` or `int` to `size_t` are no longer available during overload resolution to figure out what constructor should be called. The result is compiler errors for both `b4` and `b5`. Note that `b1`, `b2`, and `b3` are still valid definitions even if the constructors are explicit.

The only way to fix the problem, in this case, is to provide an explicit cast from `char` or `int` to `string_buffer`:

```
string_buffer b4 = string_buffer('a');
string_buffer b5 = static_cast<string_buffer>(MaxSize);
string_buffer b6 = string_buffer{ "a" };
```

With explicit constructors, the compiler is able to immediately flag erroneous situations and developers can react accordingly, either fixing the initialization with a correct value or providing an explicit cast.

This is only the case when initialization is done with copy initialization and not when using the functional or universal initialization.

The following definitions are still possible (and wrong) with explicit constructors:

```
string_buffer b7{ 'a' };
string_buffer b8('a');
```

Similar to constructors, conversion operators can be declared explicit (as shown earlier). In this case, the implicit conversions from the object type to the type specified by the conversion operator are no longer possible and require an explicit cast. Considering `b1` and `b2`, the `string_buffer` objects defined earlier, the following are no longer possible with explicit conversion `operator bool`:

```
std::cout << b1 + b2 << std::endl;
if(b1 == b2) {}
```

Instead, they require explicit conversion to `bool`:

```
std::cout << static_cast<bool>(b1) + static_cast<bool>(b2);
if(static_cast<bool>(b1) == static_cast<bool>(b2)) {}
```

See also

- *Understanding uniform initialization*

Using unnamed namespaces instead of static globals

The larger a program the greater the chances are you could run into name collisions with file locals when your program is linked. Functions or variables that are declared in a source file and are supposed to be local to the translation unit may collide with other similar functions or variables declared in another translation unit. That is because all symbols that are not declared static have external linkage and their names must be unique throughout the program. The typical C solution for this problem is to declare those symbols static, changing their linkage from external to internal and therefore making them local to a translation unit. In this recipe, we will look at the C++ solution for this problem.

Getting ready

In this recipe, we will discuss concepts such as global functions, static functions, and variables, namespaces, and translation units. Apart from these, it is required that you understand the difference between internal and external linkage; that is key for this recipe.

How to do it…

When you are in a situation where you need to declare global symbols as statics to avoid linkage problems, prefer to use unnamed namespaces:

1. Declare a namespace without a name in your source file.
2. Put the definition of the global function or variable in the unnamed namespace without making them `static`.

The following example shows two functions called `print()` in two different translation units; each of them is defined in an unnamed namespace:

```
// file1.cpp
namespace
{
  void print(std::string message)
  {
    std::cout << "[file1] " << message << std::endl;
  }
}

void file1_run()
{
  print("run");
}

// file2.cpp
namespace
{
  void print(std::string message)
  {
    std::cout << "[file2] " << message << std::endl;
  }
}

void file2_run()
{
  print("run");
}
```

How it works…

When a function is declared in a translation unit, it has external linkage. That means two functions with the same name from two different translation units would generate a linkage error because it is not possible to have two symbols with the same name. The way this problem is solved in C, and by some in C++ also, is to declare the function or variable static and change its linkage from external to internal. In this case, its name is no longer exported outside the translation unit, and the linkage problem is avoided.

The proper solution in C++ is to use unnamed namespaces. When you define a namespace like the ones shown above, the compiler transforms it to the following:

```
// file1.cpp
namespace _unique_name_ {}
using namespace _unique_name_;
namespace _unique_name_
{
  void print(std::string message)
  {
    std::cout << "[file1] " << message << std::endl;
  }
}

void file1_run()
{
  print("run");
}
```

First of all, it declares a namespace with a unique name (what the name is and how it generates that name is a compiler implementation detail and should not be a concern). At this point, the namespace is empty, and the purpose of this line is to basically establish the namespace. Second, a using directive brings everything from the `_unique_name_` namespace into the current namespace. Third, the namespace, with the compiler-generated name, is defined as it was in the original source code (when it had no name).

By defining the translation unit local `print()` functions in an unnamed namespace, they have local visibility only, yet their external linkage no longer produces linkage errors since they now have external unique names.

Unnamed namespaces are also working in a perhaps more obscure situation involving templates. Template arguments cannot be names with internal linkage, so using static variables is not possible. On the other hand, symbols in an unnamed namespace have external linkage and can be used as template arguments. This problem is shown in the following example where declaring `t1` produces a compiler error because the non-type argument expression has internal linkage. However, `t2` is correct because `Size2` has external linkage:

```
template <int const& Size>
class test {};

static int Size1 = 10;

namespace
{
  int Size2 = 10;
}
```

```
    test<Size1> t1;
    test<Size2> t2;
```

See also

- *Using inline namespaces for symbol versioning*

Using inline namespaces for symbol versioning

The C++11 standard has introduced a new type of namespace called *inline namespaces* that are basically a mechanism that makes declarations from a nested namespace look and act like they were part of the surrounding namespace. Inline namespaces are declared using the `inline` keyword in the namespace declaration (unnamed namespaces can also be inlined). This is a helpful feature for library versioning, and in this recipe, we will see how inline namespaces can be used for versioning symbols. From this recipe, you will learn how to version your source code using inline namespaces and conditional compilation.

Getting ready

In this recipe, we will discuss namespaces and nested namespaces, templates and template specializations, and conditional compilation using preprocessor macros. Familiarity with these concepts is required in order to proceed with the recipe.

How to do it…

To provide multiple versions of a library and let the user decide what version to use, do the following:

- Define the content of the library inside a namespace.
- Define each version of the library or parts of it inside an inner inline namespace.
- Use preprocessor macros and `#if` directives to enable a particular version of the library.

The following example shows a library that has two versions that clients can use:

```
namespace modernlib
{
  #ifndef LIB_VERSION_2
  inline namespace version_1
  {
    template<typename T>
    int test(T value) { return 1; }
  }
  #endif

  #ifdef LIB_VERSION_2
  inline namespace version_2
  {
    template<typename T>
    int test(T value) { return 2; }
  }
  #endif
}
```

How it works…

A member of an inline namespace is treated as if it was a member of the surrounding namespace. Such a member can be partially specialized, explicitly instantiated, or explicitly specialized. This is a transitive property, which means that if a namespace A contains an inline namespace B that contains an inline namespace C, then the members of C appear as they were members of both B and A and the members of B appear as they were members of A.

To better understand why inline namespaces are helpful, let's consider the case of developing a library that evolves over time from a first version to a second version (and further on). This library defines all its types and functions under a namespace called `modernlib`. In the first version, this library could look like this:

```
namespace modernlib
{
  template<typename T>
  int test(T value) { return 1; }
}
```

A client of the library can make the following call and get back the value 1:

```
auto x = modernlib::test(42);
```

However, the client might decide to specialize the template function `test()` as the following:

```
struct foo { int a; };

namespace modernlib
{
  template<>
  int test(foo value) { return value.a; }
}

auto y = modernlib::test(foo{ 42 });
```

In this case, the value of `y` is no longer 1, but 42 because the user-specialized function gets called.

Everything is working correctly so far, but as a library developer you decide to create a second version of the library, yet still ship both the first and the second version and let the user control what to use with a macro. In this second version, you provide a new implementation of the `test()` function that no longer returns 1, but 2. To be able to provide both the first and second implementations, you put them in nested namespaces called `version_1` and `version_2` and conditionally compile the library using preprocessor macros:

```
namespace modernlib
{
  namespace version_1
  {
    template<typename T>
    int test(T value) { return 1; }
  }

  #ifndef LIB_VERSION_2
  using namespace version_1;
  #endif
```

```
    namespace version_2
    {
      template<typename T>
      int test(T value) { return 2; }
    }

    #ifdef LIB_VERSION_2
    using namespace version_2;
    #endif
  }
```

Suddenly, the client code will break, regardless of whether it uses the first or second version of the library. That is because the test function is now inside a nested namespace, and the specialization for `foo` is done in the `modernlib` namespace, when it should actually be done in `modernlib::version_1` or `modernlib::version_2`. This is because the specialization of a template is required to be done in the same namespace where the template was declared. In this case, the client needs to change the code like this:

```
#define LIB_VERSION_2

#include "modernlib.h"

struct foo { int a; };

namespace modernlib
{
  namespace version_2
  {
    template<>
    int test(foo value) { return value.a; }
  }
}
```

This is a problem because the library leaks implementation details, and the client needs to be aware of those in order do the template specialization. These internal details are hidden with inline namespaces in the manner shown in the *How to do it…* section of this recipe. With that definition of the `modernlib` library, the client code with the specialization of the `test()` function in the `modernlib` namespace is no longer broken, because either `version_1::test()` or `version_2::test()` (depending on what version the client is actually using) acts as being part of the enclosing `modernlib` namespace when template specialization is done. The details of the implementation are now hidden to the client that only sees the surrounding namespace `modernlib`.

However, you should keep in mind that:

- The namespace `std` is reserved for the standard and should never be inlined.
- A namespace should not be defined inline if it was not inline in its first definition.

See also

- *Using unnamed namespaces instead of static globals*
- *Conditionally compiling your source code* recipe of Chapter 4, *Preprocessor and Compilation*

Using structured bindings to handle multi-return values

Returning multiple values from a function is something very common, yet there is no first-class solution in C++ to enable it directly. Developers have to choose between returning multiple values through reference parameters to a function, defining a structure to contain the multiple values or returning a `std::pair` or `std::tuple`. The first two use named variables that have the advantage that they clearly indicate the meaning of the return value, but have the disadvantage that they have to be explicitly defined. `std::pair` has its members called `first` and `second`, and `std::tuple` has unnamed members that can only be retrieved with a function call, but can be copied to named variables using `std::tie()`. None of these solutions is ideal.

C++17 extends the semantic use of `std::tie()` into a first-class core language feature that enables unpacking the values of a tuple to named variables. This feature is called *structured bindings*.

Getting ready

For this recipe, you should be familiar with the standard utility types `std::pair` and `std::tuple` and the utility function `std::tie()`.

How to do it…

To return multiple values from a function using a compiler that supports C++17 you should do the following:

1. Use an `std::tuple` for the return type.

```
std::tuple<int, std::string, double> find()
{
  return std::make_tuple(1, "marius", 1234.5);
}
```

2. Use structured bindings to unpack the values of the tuple into named objects.

```
auto [id, name, score] = find();
```

3. Use decomposition declaration to bind the returned values to variables inside an `if` statement or `switch` statement.

```
if (auto [id, name, score] = find(); score > 1000)
{
  std::cout << name << std::endl;
}
```

How it works…

Structured bindings are a language feature that works just like `std::tie()`, except that we don't have to define named variables for each value that needs to be unpacked explicitly with `std::tie()`. With structured bindings, we define all named variables in a single definition using the `auto` specifier so that the compiler can infer the correct type for each variable.

To exemplify this, let's consider the case of inserting items in a `std::map`. The insert method returns a `std::pair` containing an iterator to the inserted element or the element that prevented the insertion, and a boolean indicating whether the insertion was successful or not. The following code is very explicit and the use of `second` or `first->second` makes the code harder to read because you need to constantly figure out what they represent:

```
std::map<int, std::string> m;

auto result = m.insert({ 1, "one" });
std::cout << "inserted = " << result.second << std::endl
          << "value = " << result.first->second << std::endl;
```

The preceding code can be made more readable with the use of `std::tie`, that unpacks tuples into individual objects (and works with `std::pair` because `std::tuple` has a converting assignment from `std::pair`):

```
std::map<int, std::string> m;
std::map<int, std::string>::iterator it;
bool inserted;

std::tie(it, inserted) = m.insert({ 1, "one" });
std::cout << "inserted = " << inserted << std::endl
          << "value = " << it->second << std::endl;

std::tie(it, inserted) = m.insert({ 1, "two" });
std::cout << "inserted = " << inserted << std::endl
          << "value = " << it->second << std::endl;
```

The code is not necessarily simpler because it requires defining in advance the objects that the pair is unpacked to. Similarly, the more elements the tuple has the more objects you need to define, but using named objects makes the code easier to read.

C++17 structured bindings elevate the unpacking of tuple elements into named objects to the rank of a language feature; it does not require the use of `std::tie()`, and objects are initialized when declared:

```
std::map<int, std::string> m;
{
  auto[it, inserted] = m.insert({ 1, "one" });
  std::cout << "inserted = " << inserted << std::endl
            << "value = " << it->second << std::endl;
}

{
  auto[it, inserted] = m.insert({ 1, "two" });
  std::cout << "inserted = " << inserted << std::endl
            << "value = " << it->second << std::endl;
}
```

The use of multiple blocks in the above example is necessary because variables cannot be redeclared in the same block, and structured bindings imply a declaration using the `auto`

specifier. Therefore, if you need to make multiple calls like in the example above and use structured bindings you must either use different variable names or multiple blocks as shown above. An alternative to that is to avoid structured bindings and use `std::tie()`, because it can be called multiple times with the same variables, therefore you only need to declare them once.

In C++17, it is also possible to declare variables in `if` and `switch` statements with the form `if(init; condition)` and `switch(init; condition)`. This could be combined with structured bindings to produce simpler code. In the following example, we attempt to insert a new value into a map. The result of the call is unpacked into two variables, `it` and `inserted`, defined in the scope of the `if` statement in the initialization part. The condition of the `if` statement is evaluated from the value of the inserted object:

```
    if(auto [it, inserted] = m.insert({ 1, "two" }); inserted)
    { std::cout << it->second << std::endl; }
```

Working with Numbers and Strings

The recipes included in this chapter are as follows:

- Converting between numeric and string types
- Limits and other properties of numeric types
- Generating pseudo-random numbers
- Initializing all bits of internal state of a pseudo-random number generator
- Using raw string literals to avoid escaping characters
- Creating cooked user-defined literals
- Creating raw user-defined literals
- Creating a library of string helpers
- Verifying the format of a string using regular expressions
- Parsing the content of a string using regular expressions
- Replacing the content of a string using regular expressions
- Using string_view instead of constant string references

Introduction

Numbers and strings are the fundamental types of any programming language; all other types are based or composed of these ones. Developers are confronted all the time with tasks, such as converting between numbers and strings, parsing strings, or generating random numbers. This chapter is focused on providing useful recipes for these common tasks using modern C++ language and library features.

Converting between numeric and string types

Converting between number and string types is a ubiquitous operation. Prior to C++11, there was little support for converting numbers to strings and back, and developers had to resort mostly to type-unsafe functions and usually wrote their own utility functions in order to avoid writing the same code over and over again. With C++11, the standard library provides utility functions for converting between numbers and strings. In this recipe, you will learn how to convert between numbers and strings and the other way around using modern C++ standard functions.

Getting ready

All the utility functions mentioned in this recipe are available in the `<string>` header.

How to do it…

Use the following standard conversion functions when you need to convert between numbers and strings:

- To convert from an integer or floating point type to a string type, use `std::to_string()` or `std::to_wstring()` as shown in the following code snippet:

```
auto si = std::to_string(42);      // si="42"
auto sl = std::to_string(42l);     // sl="42"
auto su = std::to_string(42u);     // su="42"
auto sd = std::to_wstring(42.0);   // sd=L"42.000000"
auto sld = std::to_wstring(42.0l); // sld=L"42.000000"
```

- To convert from a string type to an integer type, use `std::stoi()`, `std::stol()`, `std::stoll()`, `std::stoul()`, or `std::stoull()`; refer to the following code snippet:

```
auto i1 = std::stoi("42");                 // i1 = 42
auto i2 = std::stoi("101010", nullptr, 2); // i2 = 42
auto i3 = std::stoi("052", nullptr, 8);    // i3 = 42
auto i4 = std::stoi("0x2A", nullptr, 16);  // i4 = 42
```

- To convert from a string type to a floating point type, use `std::stof()`, `std::stod()`, or `std::stold()`, as shown in the following code snippet:

```
// d1 = 123.45000000000000
auto d1 = std::stod("123.45");
// d2 = 123.45000000000000
auto d2 = std::stod("1.2345e+2");
// d3 = 123.44999980926514
auto d3 = std::stod("0xF.6E6666p3");
```

How it works…

To convert between an integral or floating point type to a string type, you can use either the `std::to_string()` or `std::to_wstring()` function. These functions are available in the `<string>` header and have overloads for signed and unsigned integer and real types. They produce the same result as `std::sprintf()` and `std::swprintf()` would produce when called with the appropriate format specifier for each type. The following code snippet list all the overloads of these two functions.

```
std::string to_string(int value);
std::string to_string(long value);
std::string to_string(long long value);
std::string to_string(unsigned value);
std::string to_string(unsigned long value);
std::string to_string(unsigned long long value);
std::string to_string(float value);
std::string to_string(double value);
std::string to_string(long double value);
std::wstring to_wstring(int value);
std::wstring to_wstring(long value);
std::wstring to_wstring(long long value);
std::wstring to_wstring(unsigned value);
std::wstring to_wstring(unsigned long value);
std::wstring to_wstring(unsigned long long value);
std::wstring to_wstring(float value);
std::wstring to_wstring(double value);
std::wstring to_wstring(long double value);
```

When it comes to the opposite conversion, there is an entire set of functions that have the name with the format **ston** (**string to number**), where **n** stands for **i** (`integer`), **l** (`long`), **ll** (`long long`), **ul** (`unsigned long`), or **ull** (`unsigned long long`). The following listing shows all these functions, each of them with two overloads, one that takes an `std::string` and one that takes an `std::wstring` as the first parameter:

```
int stoi(const std::string& str, std::size_t* pos = 0,
         int base = 10);
int stoi(const std::wstring& str, std::size_t* pos = 0,
         int base = 10);
long stol(const std::string& str, std::size_t* pos = 0,
         int base = 10);
long stol(const std::wstring& str, std::size_t* pos = 0,
         int base = 10);
long long stoll(const std::string& str, std::size_t* pos = 0,
                int base = 10);
long long stoll(const std::wstring& str, std::size_t* pos = 0,
                int base = 10);
unsigned long stoul(const std::string& str, std::size_t* pos = 0,
                    int base = 10);
unsigned long stoul(const std::wstring& str, std::size_t* pos = 0,
                    int base = 10);
unsigned long long stoull(const std::string& str,
                          std::size_t* pos = 0, int base = 10);
unsigned long long stoull(const std::wstring& str,
                          std::size_t* pos = 0, int base = 10);
float       stof(const std::string& str, std::size_t* pos = 0);
float       stof(const std::wstring& str, std::size_t* pos = 0);
double      stod(const std::string& str, std::size_t* pos = 0);
double      stod(const std::wstring& str, std::size_t* pos = 0);
long double stold(const std::string& str, std::size_t* pos = 0);
long double stold(const std::wstring& str, std::size_t* pos = 0);
```

The way the string to integral type functions work is by discarding all white spaces before a non-whitespace character, then taking as many characters as possible to form a signed or

unsigned number (depending on the case), and then converting that to the requested integral type (`stoi()` will return an `integer`, `stoul()` will return an `unsigned long`, and so on). In all the following examples, the result is integer `42`, except for the last example where the result is `-42`:

```
auto i1 = std::stoi("42");             // i1 = 42
auto i2 = std::stoi("   42");          // i2 = 42
auto i3 = std::stoi("   42fortytwo");  // i3 = 42
auto i4 = std::stoi("+42");            // i4 = 42
auto i5 = std::stoi("-42");            // i5 = -42
```

A valid integral number may consist of the following parts:

- A sign, plus (`+`) or minus (`-`) (optional).
- Prefix `0` to indicate an octal base (optional).
- Prefix `0x` or `0X` to indicate a hexadecimal base (optional).
- A sequence of digits.

The optional prefix `0` (for octal) is applied only when the specified base is `8` or `0`. Similarly, the optional prefix `0x` or `0X` (for hexadecimal) is applied only when the specified base is `16` or `0`.

The functions that convert a string to an integer have three parameters:

- The input string.
- A pointer that when not null will receive the number of characters that were processed and that can include any leading white spaces that were discarded, the sign, and the base prefix, so it should not be confused with the number of digits the integral value has.
- A number indicating the base; by default, this is `10`.

The valid digits in the input string depend on the base. For base `2`, the only valid digits are `0` and `1`; for base `5`, they are `01234`. For base `11`, the valid digits are `0-9` and characters `A` and `a`. This continues until we reach base `36` that has valid characters `0-9`, `A-Z`, and `a-z`.

The following are more examples of strings with numbers in various bases converted to decimal integers. Again, in all cases, the result is either `42` or `-42`:

```
auto i6 = std::stoi("052", nullptr, 8);
auto i7 = std::stoi("052", nullptr, 0);
auto i8 = std::stoi("0x2A", nullptr, 16);
auto i9 = std::stoi("0x2A", nullptr, 0);
auto i10 = std::stoi("101010", nullptr, 2);
auto i11 = std::stoi("22", nullptr, 20);
auto i12 = std::stoi("-22", nullptr, 20);

auto pos = size_t{ 0 };
auto i13 = std::stoi("42", &pos);      // pos = 2
auto i14 = std::stoi("-42", &pos);     // pos = 3
auto i15 = std::stoi("  +42dec", &pos);// pos = 5
```

An important thing to note is that these conversion functions throw if the conversion fails. There are two exceptions that can be thrown:

- `std::invalid_argument`: If the conversion cannot be performed:

```
try
{
   auto i16 = std::stoi("");
}
catch (std::exception const & e)
{
   // prints "invalid stoi argument"
   std::cout << e.what() << std::endl;
}
```

- `std::out_of_range`: If the converted value is outside the range of the result type (or if the underlying function sets `errno` to `ERANGE`):

```
try
{
   // OK
   auto i17 = std::stoll("12345678901234");
   // throws std::out_of_range
   auto i18 = std::stoi("12345678901234");
}
catch (std::exception const & e)
{
   // prints "stoi argument out of range"
   std::cout << e.what() << std::endl;
}
```

The other set of functions that convert a string to a floating point type is very similar, except that they don't have a parameter for the numeric base. A valid floating point value can have different representations in the input string:

- Decimal floating point expression (optional sign, sequence of decimal digits with optional point, optional `e` or `E` followed by exponent with optional sign).
- Binary floating point expression (optional sign, `0x` or `0X` prefix, sequence of hexadecimal digits with optional point, optional `p` or `P` followed by exponent with optional sign).
- Infinity expression (optional sign followed by case insensitive `INF` or `INFINITY`).
- A non-number expression (optional sign followed by case insensitive `NAN` and possibly other alphanumeric characters).

The following are various examples of converting strings to doubles:

```
auto d1 = std::stod("123.45");         // d1 =  123.45000000000000
auto d2 = std::stod("+123.45");        // d2 =  123.45000000000000
auto d3 = std::stod("-123.45");        // d3 = -123.45000000000000
auto d4 = std::stod("  123.45");       // d4 =  123.45000000000000
auto d5 = std::stod("  -123.45abc");   // d5 = -123.45000000000000
auto d6 = std::stod("1.2345e+2");      // d6 =  123.45000000000000
auto d7 = std::stod("0xF.6E6666p3");   // d7 =  123.44999980926514

auto d8 = std::stod("INF");            // d8 = inf
auto d9 = std::stod("-infinity");      // d9 = -inf
auto d10 = std::stod("NAN");           // d10 = nan
auto d11 = std::stod("-nanabc");       // d11 = -nan
```

The floating-point base 2 scientific notation, seen earlier in the form `0xF.6E6666p3`, is not the topic of this recipe. However, for a clear understanding, a short description is provided; although, it is recommended that you see additional references for details. A floating-point constant in the base 2 scientific notation is composed of several parts:

- The hexadecimal prefix `0x`.

- An integer part, in this example was `F`, which in decimal is 15.
- A fractional part, which in this example was `6E6666`, or `01101110011001100110110` in binary. To convert that to decimal, we need to add inverse powers of two: `1/4 + 1/8 + 1/32 + 1/64 + 1/128 + ...`.
- A suffix, representing a power of 2; in this example, `p3` means 2 at the power of 3.

The value of the decimal equivalent is determined by multiplying the significant (composed of the integer and fractional parts) and the base at the power of exponent. For the given hexadecimal base 2 floating point literal, the significant is `15.4312499...` (note that digits after the seventh one are not shown), the base is 2, and the exponent is 3. Therefore, the result is `15.4212499... * 8`, which is `123.44999980926514`.

See also

- *Limits and other properties of numeric types*

Limits and other properties of numeric types

Sometimes, it is necessary to know and use the minimum and maximum values representable with a numeric type, such as `char`, `int`, or `double`. Many developers are using standard C macros for this, such as `CHAR_MIN`/`CHAR_MAX`, `INT_MIN`/`INT_MAX`, or `DBL_MIN`/`DBL_MAX`. C++ provides a class template called `numeric_limits` with specializations for every numeric type that enables you to query the minimum and maximum value of a type, but is not limited to that and offers additional constants for type properties querying, such as whether a type is signed or not, how many bits it needs for representing its values, for floating point types whether it can represent infinity, and many others. Prior to C++11, the use of `numeric_limits<T>` was limited because it could not be used in places where constants were needed (examples can include the size of arrays and switch cases). Due to that, developers preferred to use the C macros throughout their code. In C++11, that is no longer the case, as all the static members of `numeric_limits<T>` are now `constexpr`, which means they can be used everywhere a constant expression is expected.

Getting ready

The `numeric_limits<T>` class template is available in the namespace `std` in the `<limits>` header.

How to do it…

Use `std::numeric_limits<T>` to query various properties of a numeric type `T`:

- Use `min()` and `max()` static methods to get the smallest and largest finite numbers of a type:

```
template<typename T, typename I>
T minimum(I const start, I const end)
{
  T minval = std::numeric_limits<T>::max();
  for (auto i = start; i < end; ++i)
  {
    if (*i < minval)
      minval = *i;
  }
  return minval;
}

int range[std::numeric_limits<char>::max() + 1] = { 0 };

switch(get_value())
{
  case std::numeric_limits<int>::min():
  break;
}
```

- Use other static methods and static constants to retrieve other properties of a numeric type:

```
auto n = 42;
std::bitset<std::numeric_limits<decltype(n)>::digits>
  bits { static_cast<unsigned long long>(n) };
```

In C++11, there is no limitation to where `std::numeric_limits<T>` can be used; therefore, preferably use it over C macros in your modern C++ code.

How it works…

The `std::numeric_limits<T>` is a class template that enables developers to query property of numeric types. Actual values are available through specializations, and the standard library provides specializations for all the built-in numeric types (`char`, `short`, `int`, `long`, `float`, `double`, and so on). In addition, third parties may provide additional implementation for other types. An example could be a numeric library that implements a `bigint` integer type and a `decimal` type and provides specializations of `numeric_limits` for these types (such as `numeric_limits<bigint>` and `numeric_limits<decimal>`).

The following specializations of numeric types are available in the `<limits>` header. Note that specializations for `char16_t` and `char32_t` are new in C++11; the others were available previously. Apart from the specializations listed ahead, the library also includes specializations for every `cv-qualified` version of these numeric types, and they are identical to the unqualified specialization. For example, consider type `int`; there are four actual specializations (and they are identical): `numeric_limits<int>`, `numeric_limits<const int>`, `numeric_limits<volatile int>`, and `numeric_limits<const volatile int>`:

```
template<> class numeric_limits<bool>;
template<> class numeric_limits<char>;
template<> class numeric_limits<signed char>;
template<> class numeric_limits<unsigned char>;
template<> class numeric_limits<wchar_t>;
template<> class numeric_limits<char16_t>;
template<> class numeric_limits<char32_t>;
template<> class numeric_limits<short>;
template<> class numeric_limits<unsigned short>;
template<> class numeric_limits<int>;
template<> class numeric_limits<unsigned int>;
template<> class numeric_limits<long>;
template<> class numeric_limits<unsigned long>;
template<> class numeric_limits<long long>;
template<> class numeric_limits<unsigned long long>;
template<> class numeric_limits<float>;
template<> class numeric_limits<double>;
template<> class numeric_limits<long double>;
```

As mentioned earlier, in C++11, all static members of `numeric_limits` are `constexpr`, which means they can be used in all places where constant expressions are needed. These have several major advantages over C++ macros:

- They are easier to remember, as the only thing you need to know is the name of the type that you should know anyway, and not countless names of macros.
- They support types that are not available in C, such as `char16_t` and `char32_t`.
- They are the only possible solution for templates where you don’t know the type.
- Minimum and maximum are only two of the various properties of types it provides; therefore, its actual use is beyond the numeric limits. As a side note, for this reason, the class should have been perhaps called `numeric_properties`, instead of `numeric_limits`.

The following function template `print_type_properties()` prints the minimum and maximum finite values of the type as well as other information:

```
template <typename T>
void print_type_properties()
```

```
    {
      std::cout
        << "min="
        << std::numeric_limits<T>::min()        << std::endl
        << "max="
        << std::numeric_limits<T>::max()        << std::endl
        << "bits="
        << std::numeric_limits<T>::digits       << std::endl
        << "decdigits="
        << std::numeric_limits<T>::digits10     << std::endl
        << "integral="
        << std::numeric_limits<T>::is_integer   << std::endl
        << "signed="
        << std::numeric_limits<T>::is_signed    << std::endl
        << "exact="
        << std::numeric_limits<T>::is_exact     << std::endl
        << "infinity="
        << std::numeric_limits<T>::has_infinity << std::endl;
    }
```

If we call the `print_type_properties()` function for unsigned `short`, `int`, and `double`, it will have the following output:

`unsigned short`	`int`	`double`
min=0	min=-2147483648	min=2.22507e-308
max=65535	max=2147483647	max=1.79769e+308
bits=16	bits=31	bits=53
decdigits=4	decdigits=9	decdigits=15
integral=1	integral=1	integral=0
signed=0	signed=1	signed=1
exact=1	exact=1	exact=0
infinity=0	infinity=0	infinity=1

The one thing to take note of is the difference between the `digits` and `digits10` constants:

- `digits` represent the number of bits (excluding the sign bit if present) and padding bits (if any) for integral types and the number of bits of the mantissa for floating point types.
- `digits10` is the number of decimal digits that can be represented by a type without a change. To understand this better, let's consider the case of `unsigned short`. This is a 16-bit integral type. It can represent numbers between 0 and 65536. It can represent numbers up to five decimal digits, 10,000 to 65,536, but it cannot represent all five decimal digit numbers, as numbers from 65,537 to 99,999 require more bits. Therefore, the largest numbers that it can represent without requiring more bits have four decimal digits (numbers from 1,000 to 9,999). This is the value indicated by `digits10`. For integral types, it has a direct relationship to constant `digits`; for an integral type `T`, the value of `digits10` is `std::numeric_limits<T>::digits * std::log10(2)`.

Generating pseudo-random numbers

Generating random numbers is necessary for a large variety of applications, from games to cryptography, from sampling to forecasting. However, the term *random numbers* is not actually correct, as the generation of numbers through mathematical formulas is deterministic and does not produce true random numbers, but numbers that look random and are called *pseudo-random*. True randomness can only be achieved through hardware devices, based on physical processes, and even that can be challenged, as one may consider even the universe to be actually deterministic. Modern C++ provides support for generating pseudo-random numbers through a pseudo-random number library containing number generators and distributions. Theoretically, it can also produce true random numbers, but in practice, those could actually be only pseudo-random.

Getting ready

In this recipe, we discuss the standard support for generating pseudo-random numbers. Understanding the difference between random and pseudo-random numbers is the key. On the other hand, being familiar with various statistical distributions is a plus. It is mandatory, though, that you know what a uniform distribution is because all engines in the library produce numbers that are uniformly distributed.

How to do it…

To generate pseudo-random numbers in your application, you should perform the following steps:

1. Include the header `<random>`:

```
#include <random>
```

2. Use an `std::random_device` generator for seeding a pseudo-random engine:

```
std::random_device rd{};
```

3. Use one of the available engines for generating numbers and initialize it with a random seed:

```
auto mtgen = std::mt19937{ rd() };
```

4. Use one of the available distributions for converting the output of the engine to one of the desired statistical distributions:

```
auto ud = std::uniform_int_distribution<>{ 1, 6 };
```

5. Generate the pseudo-random numbers:

```
for(auto i = 0; i < 20; ++i)
  auto number = ud(mtgen);
```

How it works…

The pseudo-random number library contains two types of components:

- *Engines*, which are generators of random numbers; these could produce either pseudo-random numbers with a uniform distribution or, if available, actual random numbers.
- *Distributions* that convert the output of an engine into a statistical distribution.

All engines (except for `random_device`) produce integer numbers in a uniform distribution, and all engines implement the following methods:

- `min()`: This is a static method that returns the minimum value that can be produced by the generator.
- `max()`: This is a static method that returns the maximum value that can be produced by the generator.
- `seed()`: This initializes the algorithm with a start value (except for `random_device`, which cannot be seeded).
- `operator()`: This generates a new number uniformly distributed between `min()` and `max()`.
- `discard()`: This generates and discards a given number of pseudo-random numbers.

The following engines are available:

- `linear_congruential_engine`: This is a linear congruential generator that produces numbers using the following formula:

$$x(i) = (A * x(i-1) + C) \bmod M$$

- `mersenne_twister_engine`: This is a Mersenne twister generator that keeps a value on $W * (N-1) * R$ bits; each time a number needs to be generated, it extracts W bits. When all bits have been used, it twists the large value by shifting and mixing the bits so that it has a new set of bits to extract from.
- `subtract_with_carry_engine`: This is a generator that implements a *subtract with carry* algorithm based on the following formula:

$$x(i) = (x(i - R) - x(i - S) - cy(i - 1)) \bmod M$$

In the preceding formula, *cy* is defined as:

$$cy(i) = x(i - S) - x(i - R) - cy(i - 1) < 0 \ ? \ 1 : 0$$

In addition, the library provides engine adapters that are also engines wrapping another engine and producing numbers based on the output of the base engine. Engine adapters implement the same methods mentioned earlier for the base engines. The following engine adapters are available:

- `discard_block_engine`: A generator that from every block of P numbers generated by the base engine keeps only R numbers, discarding the rest.
- `independent_bits_engine`: A generator that produces numbers with a different number of bits than the base engine.
- `shuffle_order_engine`: A generator that keeps a shuffled table of K numbers produced by the base engine and returns numbers from this table, replacing them with numbers generated by the base engine.

All these engines and engine adaptors are producing pseudo-random numbers. The library, however, provides another engine called `random_device` that is supposed to produce non-deterministic numbers, but this is not an actual constraint as physical sources of random entropy might not be available. Therefore, implementations of `random_device` could actually be based on a pseudo-random engine. The `random_device` class cannot be seeded like the other engines and has an additional method called `entropy()` that returns the random device entropy, which is 0 for a deterministic generator and nonzero for a non-deterministic generator. However, this is not a reliable method for determining whether the device is actually deterministic or non-deterministic. For instance, both GNU `libstdc++` and LLVM `libc++` implement a non-deterministic device, but return `0` for entropy. On the other hand, `VC++` and `boost.random` return `32` and `10`, respectively, for entropy.

All these generators produce integers in a uniform distribution. This is, however, only one of the many possible statistical distributions that random numbers are needed in most applications. To be able to produce numbers (either integer or real) in other distributions, the library provides several classes that are called *distributions* and are converting the output of an engine according to the statistical distribution it implements. The following distributions are available:

Type	**Class name**	**Numbers**	**Statistical distribution**
Uniform	`uniform_int_distribution`	integer	Uniform
	`uniform_real_distribution`	real	Uniform
Bernoulli	`bernoulli_distribution`	boolean	Bernoulli
	`binomial_distribution`	integer	binomial
	`negative_binomial_distribution`	integer	negative binomial
	`geometric_distribution`	integer	geometric
Poisson	`poisson_distribution`	integer	poisson
	`exponential_distribution`	real	exponential
	`gamma_distribution`	real	gamma
	`weibull_distribution`	real	Weibull
	`extreme_value_distribution`	real	extreme value
Normal	`normal_distribution`	real	standard normal (Gaussian)

	`lognormal_distribution`	real	lognormal
	`chi_squared_distribution`	real	chi-squared
	`cauchy_distribution`	real	Cauchy
	`fisher_f_distribution`	real	Fisher’s F-distribution
	`student_t_distribution`	real	Student’s t-distribution
Sampling	`discrete_distribution`	integer	discrete
	`piecewise_constant_distribution`	real	values distributed on constant subintervals
	`piecewise_linear_distribution`	real	values distributed on defined subintervals

Each of the engines provided by the library has advantages and disadvantages. The linear congruential engine has a small internal state, but it is not very fast. On the other hand, the subtract with carry engine is very fast, but requires more memory for its internal state. The Mersenne twister is the slowest of them and the one that has the largest internal state, but when initialized appropriately can produce the longest non-repeating sequence of numbers. In the following examples, we will use `std::mt19937`, a 32-bit Mersenne twister with 19,937 bits of internal state.

The simplest way to generate random numbers looks like this:

```
auto mtgen = std::mt19937 {};
for (auto i = 0; i < 10; ++i)
  std::cout << mtgen() << std::endl;
```

In this example, `mtgen` is an `std::mt19937` Mersenne twister. To generate numbers, you only need to use the call operator that advances the internal state and returns the next pseudo-random number. However, this code is flawed, as the engine is not seeded. As a result, it always produces the same sequence of numbers, which is probably not what you want in most cases.

There are different approaches for initializing the engine. One approach, common with the C rand library, is to use the current time. In modern C++, it should look like this:

```
auto seed = std::chrono::high_resolution_clock::now()
            .time_since_epoch()
            .count();
auto mtgen = std::mt19937{ static_cast<unsigned int>(seed) };
```

In this example, `seed` is a number representing the number of ticks since the clock’s epoch until the present moment. This number is then used to seed the engine. The problem with this approach is that the value of that `seed` is actually deterministic, and in some classes of applications it could be prone to attacks. A more reliable approach is to seed the generator with actual random numbers. The `std::random_device` class is an engine that is supposed to return true random numbers, though implementations could actually be based on a pseudo-random generator:

```
std::random_device rd;
auto mtgen = std::mt19937 {rd()};
```

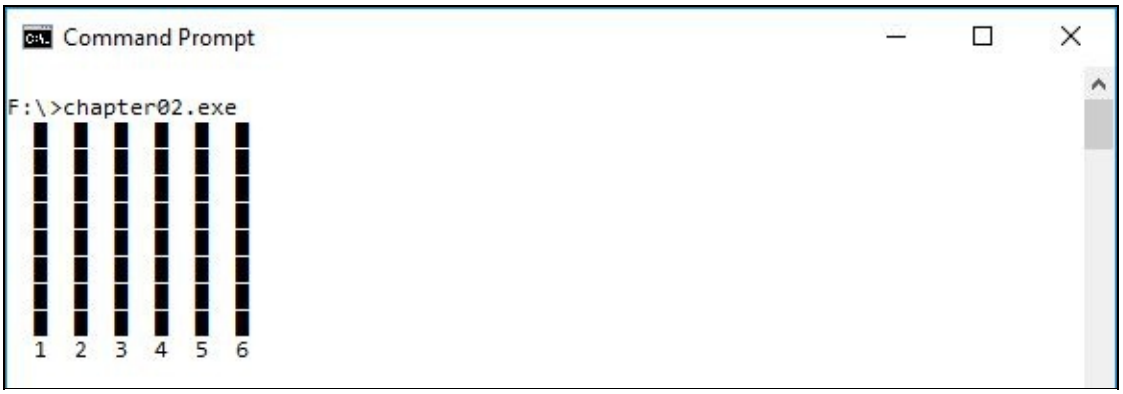
Numbers produced by all engines follow a uniform distribution. To convert the result to another statistical distribution, we have to use a distribution class. To show how generated numbers are distributed according to the selected distribution, we will use the following function. This function generates a specified number of pseudo-random numbers and counts their repetition in a map. The values from the map are then used to produce a bar-like diagram showing how often each number occurred:

```
void generate_and_print(
  std::function<int(void)> gen,
  int const iterations = 10000)
{
  // map to store the numbers and their repetition
  auto data = std::map<int, int>{};

  // generate random numbers
  for (auto n = 0; n < iterations; ++n)
    ++data[gen()];

  // find the element with the most repetitions
  auto max = std::max_element(
             std::begin(data), std::end(data),
             [](auto kvp1, auto kvp2) {
    return kvp1.second < kvp2.second; });

  // print the bars
  for (auto i = max->second / 200; i > 0; --i)
  {
    for (auto kvp : data)
    {
      std::cout
        << std::fixed << std::setprecision(1) << std::setw(3)
        << (kvp.second / 200 >= i ? (char)219 : ' ');
    }

    std::cout << std::endl;
  }

  // print the numbers
  for (auto kvp : data)
  {
    std::cout
      << std::fixed << std::setprecision(1) << std::setw(3)
      << kvp.first;
  }

  std::cout << std::endl;
}
```

The following code generates random numbers using the `std::mt19937` engine with a uniform distribution in the range `[1, 6]`; that is basically what you get when you throw a dice:

```
std::random_device rd{};
auto mtgen = std::mt19937{ rd() };
auto ud = std::uniform_int_distribution<>{ 1, 6 };
generate_and_print([&mtgen, &ud]() {return ud(mtgen); });
```

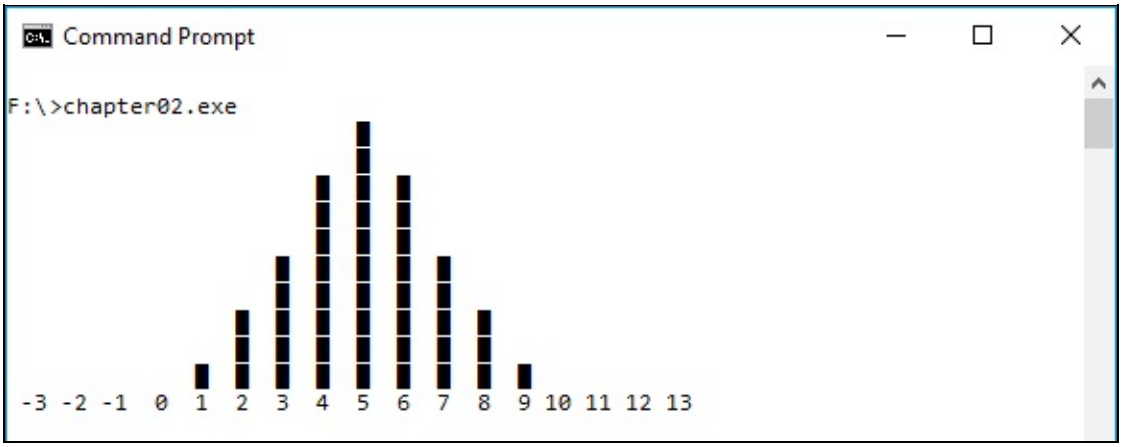
The output of the program looks like this:



In the next and final example, we change the distribution to a normal distribution with the mean `5` and the standard deviation `2`. This distribution produces real numbers; therefore, in order to use the previous `generate_and_print()` function, the numbers must be rounded to integers:

```
std::random_device rd{};
auto mtgen = std::mt19937{ rd() };
auto nd = std::normal_distribution<>{ 5, 2 };

generate_and_print(
  [&mtgen, &nd]() {
    return static_cast<int>(std::round(nd(mtgen))); });
```

The following will be the output of the earlier code:



See also

- *Initializing all bits of internal state of a pseudo-random number generator*

Initializing all bits of internal state of a pseudo-random number generator

In the previous recipe, we have looked at the pseudo-random number library with its components and how it can be used to produce numbers in different statistical distributions. One important factor that was overlooked in that recipe is the proper initialization of the pseudo-random number generators. In this recipe, you will learn how to initialize a generator in order to produce the best sequence of pseudo-random numbers.

Getting ready

You should read the previous recipe, *Generating pseudo-random numbers*, to get an overview of what the pseudo-random number library offers.

How to do it…

To properly initialize a pseudo-random number generator to produce the best sequence of pseudo-random numbers, perform the following steps:

1. Use an `std::random_device` to produce random numbers to be used as seeding values:

```
std::random_device rd;
```

2. Generate random data for all internal bits of the engine:

```
std::array<int, std::mt19937::state_size> seed_data {};
std::generate(std::begin(seed_data), std::end(seed_data),
              std::ref(rd));
```

3. Create an `std::seed_seq` object from the previously generated pseudo-random data:

```
std::seed_seq seq(std::begin(seed_data), std::end(seed_data));
```

4. Create an engine object and initialize all the bits representing the internal state of the engine; for example, a `mt19937` has 19,937 bits of internal states:

```
auto eng = std::mt19937{ seq };
```

5. Use the appropriate distribution based on the requirements of the application:

```
auto dist = std::uniform_real_distribution<>{ 0, 1 };
```

How it works…

In all examples shown in the previous recipe, we used an `std::mt19937` engine to produce pseudo-random numbers. Though the Mersenne twister is slower than the other engines, it can produce the longest sequences of non-repeating numbers and with the best spectral characteristics. However, initializing the engine in the manner shown in the previous recipe will not have this effect. With a careful analysis (that is beyond the purpose of this recipe or this book), it can be shown that the engine has a bias toward producing some values repeatedly and omitting others, thus generating numbers not in a uniform distribution, but rather in a binomial or Poisson distribution. The problem is that the internal state of `mt19937` has 624 32-bit integers, and in the examples from the previous recipe we have only initialized one of them.

When working with the pseudo-random number library, remember the following rule of thumb (shown in the information box):

In order to produce the best results, engines must have all their internal state properly initialized before generating numbers.

The pseudo-random number library provides a class for this particular purpose, called `std::seed_seq`. This is a generator that can be seeded with any number of 32-bit integers and produces a requested number of integers evenly distributed in the 32-bit space.

In the preceding code from the *How to do it…* section, we defined an array called `seed_data` with a number of 32-bit integers equal to the internal state of the `mt19937` generator; that is 624 integers. Then, we initialized the array with random numbers produced by an `std::random_device`. The array was later used to seed an `std::seed_seq`, which in turn was used to seed the `mt19937` generator.

Creating cooked user-defined literals

Literals are constants of built-in types (numerical, boolean, character, character string, and pointer) that cannot be altered in a program. The language defines a series of prefixes and suffixes to specify literals (and the prefix/suffix is actually part of the literal). C++11 allows creating user-defined literals by defining functions called *literal operators* that introduce suffixes for specifying literals. These work only with numerical character and character string types. This opens the possibility of defining both standard literals in future versions and allows developers to create their own literals. In this recipe, we will see how we can create our own cooked literals.

Getting ready

User-defined literals can have two forms: *raw* and *cooked*. Raw literals are not processed by the compiler, whereas cooked literals are values processed by the compiler (examples can include handling escape sequences in a character string or identifying numerical values such as integer 2898 from literal 0xBAD). Raw literals are only available for integral and floating-point types, whereas cooked literals are also available for character and character string literals.

How to do it…

To create cooked user-defined literals, you should follow these steps:

1. Define your literals in a separate namespace to avoid name clashes.
2. Always prefix the user-defined suffix with an underscore (_).
3. Define a literal operator of the following form for cooked literals:

```
T operator "" _suffix(unsigned long long int);
T operator "" _suffix(long double);
T operator "" _suffix(char);
T operator "" _suffix(wchar_t);
T operator "" _suffix(char16_t);
T operator "" _suffix(char32_t);
T operator "" _suffix(char const *, std::size_t);
T operator "" _suffix(wchar_t const *, std::size_t);
T operator "" _suffix(char16_t const *, std::size_t);
T operator "" _suffix(char32_t const *, std::size_t);
```

The following example creates a user-defined literal for specifying kilobytes:

```
namespace compunits
{
  constexpr size_t operator "" _KB(unsigned long long const size)
  {
    return static_cast<size_t>(size * 1024);
  }
}

auto size{ 4_KB };         // size_t size = 4096;

using byte = unsigned char;
auto buffer = std::array<byte, 1_KB>{};
```

How it works…

When the compiler encounters a user-defined literal with a user-defined suffix `s` (it always has a leading underscore for third-party suffixes, as the suffixes without a leading underscore are reserved for the standard library) it does an unqualified name lookup in order to identify a function with the name operator `"operator "" s`. If it finds one, then it calls it according to the type of the literal and the type of the literal operator. Otherwise, the compiler will yield and error.

In the example from the *How to do it…* section, the literal operator is called `operator "" _KB` and has an argument of type `unsigned long long int`. This is the only integral type possible for literal operators for handling integral types. Similarly, for floating-point user-defined literals, the parameter type must be `long double` since for numeric types the literal operators must be able to handle the largest possible values. This literal operator returns a `constexpr` value so that it can be used where compile time values are expected, such as specifying the size of an array as shown in the above example.

When the compiler identifies a user-defined literal and has to call the appropriate user-defined literal operator, it will pick the overload from the overload set according to the following rules:

- **For integral literals**: It calls in the following order: the operator that takes an `unsigned long long`, the raw literal operator that takes a `const char*`, or the literal operator template.
- **For floating-point literals**: It calls in the following order: the operator that takes a `long double`, the raw literal operator that takes a `const char*`, or the literal operator template.
- **For character literals**: It calls the appropriate operator depending on the character type (`char`, `wchar_t`, `char16_t`, and `char32_t`).
- **For string literals**: It calls the appropriate operator, depending on the string type that takes a pointer to the string of characters and the size.

In the following example, we define a system of units and quantities. We want to operate with kilograms, pieces, liters, and other types of units. This could be useful in a system that can process orders and you need to specify the amount and unit for each article. The following are defined in the namespace `units`:

- A scoped enumeration for the possible types of units (kilogram, meter, liter, and pieces):

```
enum class unit { kilogram, liter, meter, piece, };
```

- A class template to specify quantities of a particular unit (such as 3.5 kilograms or 42 pieces):

```
template <unit U>
```

```
class quantity
{
  const double amount;
  public:
    constexpr explicit quantity(double const a) :
      amount(a) {}

  explicit operator double() const { return amount; }
};
```

- The `operator+` and `operator-` functions for the `quantity` class template in order to be able to add and subtract quantities:

```
template <unit U>
constexpr quantity<U> operator+(quantity<U> const &q1,
                                quantity<U> const &q2)
{
  return quantity<U>(static_cast<double>(q1) +
                     static_cast<double>(q2));
}

template <unit U>
constexpr quantity<U> operator-(quantity<U> const &q1,
                                quantity<U> const &q2)
{
  return quantity<U>(static_cast<double>(q1) -
                     static_cast<double>(q2));
}
```

- Literal operators to create `quantity` literals, defined in an inner namespace called `unit_literals`. The purpose of this is to avoid possible name clashes with literals from other namespaces. If such collisions do happen, developers could select the ones that they should use using the appropriate namespace in the scope where the literals need to be defined:

```
namespace unit_literals
{
  constexpr quantity<unit::kilogram> operator "" _kg(
      long double const amount)
  {
    return quantity<unit::kilogram>
      { static_cast<double>(amount) };
  }

  constexpr quantity<unit::kilogram> operator "" _kg(
     unsigned long long const amount)
  {
    return quantity<unit::kilogram>
      { static_cast<double>(amount) };
  }

  constexpr quantity<unit::liter> operator "" _l(
     long double const amount)
  {
     return quantity<unit::liter>
       { static_cast<double>(amount) };
  }

  constexpr quantity<unit::meter> operator "" _m(
     long double const amount)
  {
    return quantity<unit::meter>
      { static_cast<double>(amount) };
  }

  constexpr quantity<unit::piece> operator "" _pcs(
     unsigned long long const amount)
  {
    return quantity<unit::piece>
```

```
            { static_cast<double>(amount) };
        }
    }
```

By looking carefully, you can note that the literal operators defined earlier are not the same:

- `_kg` is defined for both integral and floating point literals; that enables us to create both integral and floating point values such as `1_kg` and `1.0_kg`.
- `_l` and `_m` are defined only for floating point literals; that means we can only define quantity literals for these units with floating points, such as `4.5_l` and `10.0_m`.
- `_pcs` is only defined for integral literals; that means we can only define quantities of an integer number of pieces, such as `42_pcs`.

Having these literal operators available, we can operate with various quantities. The following examples show both valid and invalid operations:

```
using namespace units;
using namespace unit_literals;

auto q1{ 1_kg };    // OK
auto q2{ 4.5_kg };  // OK
auto q3{ q1 + q2 }; // OK
auto q4{ q2 - q1 }; // OK

// error, cannot add meters and pieces
auto q5{ 1.0_m + 1_pcs };
// error, cannot have an integer number of liters
auto q6{ 1_l };
// error, can only have an integer number of pieces
auto q7{ 2.0_pcs}
```

`q1` is a quantity of 1 kg; that is an integer value. Since an overloaded `operator "" _kg(unsigned long long const)` exists, the literal can be correctly created from the integer 1. Similarly, `q2` is a quantity of 4.5 kilograms; that is a real value. Since an `overload operator "" _kg(long double)` exists, the literal can be created from the double floating point value 4.5.

On the other hand, `q6` is a quantity of 1 liter. Since there is no overloaded `operator "" _l(unsigned long long)`, the literal cannot be created. It would require an overload that takes a `unsigned long long`, but such an overload does not exist. Similarly, `q7` is a quantity of 2.0 pieces, but piece literals can only be created from integer values and, therefore, this generates another compiler error.

There’s more…

Though user-defined literals are available from C++11, standard literal operators have been available only from C++14. The following is a list of these standard literal operators:

- `operator""s` for defining `std::basic_string` literals:

```
using namespace std::string_literals;

auto s1{  "text"s }; // std::string
auto s2{ L"text"s }; // std::wstring
auto s3{ u"text"s }; // std::u16string
auto s4{ U"text"s }; // std::u32string
```

- `operator""h`, `operator""min`, `operator""s`, `operator""ms`, `operator""us`, and `operator""ns` for creating a `std::chrono::duration` value:

```
using namespace std::literals::chrono_literals;

// std::chrono::duration<long long>
auto timer {2h + 42min + 15s};
```

- `operator""if`, `operator""i`, and `operator""il` for creating a `std::complex` value:

```
using namespace std::literals::complex_literals;

auto c{ 12.0 + 4.5i }; // std::complex<double>
```

See also

- *Using raw string literals to avoid escaping characters*
- *Creating raw user-defined literals*

Creating raw user-defined literals

In the previous recipe, we have looked at the way C++11 allows library implementers and developers to create user-defined literals and the user-defined literals available in the C++14 standard. However, user-defined literals have two forms, a cooked form, where the literal value is processed by the compiler before being supplied to the literal operator, and a raw form, in which the literal is not parsed by the compiler. The latter is only available for integral and floating-point types. In this recipe, we will look at creating raw user-defined literals.

Getting ready

Before continuing with this recipe, it is strongly recommended that you go through the previous one, *Creating cooked user-defined literals*, as general details about user-defined literals will not be reiterated here.

To exemplify the way raw user-defined literals can be created, we will define binary literals. These binary literals can be of 8-bit, 16-bit, and 32-bit (unsigned) types. These types will be called `byte8`, `byte16`, and `byte32`, and the literals we create will be called `_b8`, `_b16`, and `_b32`.

How to do it…

To create raw user-defined literals, you should follow these steps:

1. Define your literals in a separate namespace to avoid name clashes.
2. Always prefix the used-defined suffix with an underscore (_).
3. Define a literal operator or literal operator template of the following form:

```
T operator "" _suffix(const char*);

template<char...> T operator "" _suffix();
```

The following sample shows a possible implementation of 8-bit, 16-bit, and 32-bit binary literals:

```
namespace binary
{
  using byte8  = unsigned char;
  using byte16 = unsigned short;
  using byte32 = unsigned int;

  namespace binary_literals
  {
    namespace binary_literals_internals
    {
      template <typename CharT, char... bits>
      struct binary_struct;

      template <typename CharT, char... bits>
      struct binary_struct<CharT, '0', bits...>
      {
        static constexpr CharT value{
          binary_struct<CharT, bits...>::value };
      };

      template <typename CharT, char... bits>
      struct binary_struct<CharT, '1', bits...>
      {
        static constexpr CharT value{
          static_cast<CharT>(1 << sizeof...(bits)) |
          binary_struct<CharT, bits...>::value };
      };

      template <typename CharT>
      struct binary_struct<CharT>
      {
        static constexpr CharT value{ 0 };
      };
    }

    template<char... bits>
    constexpr byte8 operator""_b8()
    {
      static_assert(
        sizeof...(bits) <= 8,
        "binary literal b8 must be up to 8 digits long");

      return binary_literals_internals::
                binary_struct<byte8, bits...>::value;
    }

    template<char... bits>
    constexpr byte16 operator""_b16()
    {
      static_assert(
        sizeof...(bits) <= 16,
        "binary literal b16 must be up to 16 digits long");
```

```
      return binary_literals_internals::
                binary_struct<byte16, bits...>::value;
    }

    template<char... bits>
    constexpr byte32 operator""_b32()
    {
      static_assert(
         sizeof...(bits) <= 32,
         "binary literal b32 must be up to 32 digits long");

      return binary_literals_internals::
                binary_struct<byte32, bits...>::value;
    }

  }
}
```

How it works…

The implementation in the previous section enables us to define binary literals of the form 1010_b8 (a `byte8` value of decimal 10) or 000010101100_b16 (a `byte16` value of decimal 2130496). However, we want to make sure that we do not exceed the number of digits for each type. In other words, values such as 111100001_b8 should be illegal and the compiler should yield an error.

First of all, we define everything inside a namespace called `binary` and start with introducing several type aliases (`byte8`, `byte16`, and `byte32`).

The literal operator templates are defined in a nested namespace called `binary_literal_internals`. This is a good practice in order to avoid name collision with other literal operators from other namespaces. Should something like that happen, you can choose to use the appropriate namespace in the right scope (such as one namespace in a function or block and another namespace in another function or block).

The three literal operator templates are very similar. The only things that are different are their names (`_b8`, `_16`, and `_b32`), return type (`byte8`, `byte16`, and `byte32`), and the condition in the static assert that checks the number of digits.

We will explore the details of variadic template and template recursion in a later recipe; however, for a better understanding, this is how this particular implementation works: `bits` is a template parameter pack, that is not a single value, but all the values the template could be instantiated with. For example, if we consider the literal `1010_b8`, then the literal operator template would be instantiated as `operator"" _b8<'1', '0', '1', '0'>()`. Before proceeding with computing the binary value, we check the number of digits in the literal. For `_b8`, this must not exceed eight (including any trailing zeros). Similarly, it should be up to 16 digits for `_b16` and 32 for `_b32`. For this, we use the `sizeof...` operator that returns the number of elements in a parameter pack (in this case, `bits`).

If the number of digits is correct, we can proceed to expand the parameter pack and recursively compute the decimal value represented by the binary literal. This is done with the help of an additional class template and its specializations. These templates are defined in yet another nested namespace, called `binary_literals_internals`. This is also a good practice because it hides (without proper qualification) the implementation details from the client (unless an explicit using namespace directive makes them available to the current namespace).

Even though this looks like recursion, it is not a true runtime recursion, because after the compiler expands and generates the code from templates, what we end up with is basically calls to overloaded functions with a different number of parameters. This is later explained in the recipe Writing a function template with a variable number of arguments.

The `binary_struct` class template has a template type `CharT` for the return type of the function (we need this because our literal operator templates should return either `byte8`, `byte16`, or

`byte32`) and a parameter pack:

```
template <typename CharT, char... bits>
struct binary_struct;
```

Several specializations of this class template are available with parameter pack decomposition. When the first digit of the pack is ‘0’, the computed value remains the same, and we continue expanding the rest of the pack. If the first digit of the pack is ‘1’, then the new value is 1 shifted to the left with the number of digits in the remainder of the pack bit, or the value of the rest of the pack:

```
template <typename CharT, char... bits>
struct binary_struct<CharT, '0', bits...>
{
  static constexpr CharT value{
    binary_struct<CharT, bits...>::value };
};

template <typename CharT, char... bits>
struct binary_struct<CharT, '1', bits...>
{
  static constexpr CharT value{
    static_cast<CharT>(1 << sizeof...(bits)) |
    binary_struct<CharT, bits...>::value };
};
```

The last specialization covers the case when the pack is empty; in this case we return 0:

```
template <typename CharT>
struct binary_struct<CharT>
{
  static constexpr CharT value{ 0 };
};
```

After defining these helper classes, we could implement the `byte8`, `byte16`, and `byte32` binary literals as intended. Note that we need to bring the content of the namespace `binary_literals` in the current namespace in order to use the literal operator templates:

```
using namespace binary;
using namespace binary_literals;
auto b1 = 1010_b8;
auto b2 = 101010101010_b16;
auto b3 = 10101010101010101010101010_b32;
```

The following definitions trigger compiler errors because the condition in `static_assert` is not met:

```
// binary literal b8 must be up to 8 digits long
auto b4 = 0011111111_b8;
// binary literal b16 must be up to 16 digits long
auto b5 = 001111111111111111_b16;
// binary literal b32 must be up to 32 digits long
auto b6 = 00111111111111111111111111111111_b32;
```

See also

- *Using raw string literals to avoid escaping character*s
- *Creating cooked user-defined literals*
- *Writing a function template with variable number of arguments* recipe of Chapter 3, *Exploring Functions*
- *Creating type aliases and alias templates* recipe of Chapter 1, *Learning Modern Core Language Features*

Using raw string literals to avoid escaping characters

Strings may contain special characters, such as non-printable characters (newline, horizontal and vertical tab, and so on), string and character delimiters (double and single quotes) or arbitrary octal, hexadecimal, or Unicode values. These special characters are introduced with an escape sequence that starts with a backslash, followed by either the character (examples include `'` and `"`), its designated letter (examples include `n` for a new line, `t` for a horizontal tab), or its value (examples include octal 050, hexadecimal XF7, or Unicode U16F0). As a result, the backslash character itself has to be escaped with another backslash character. This leads to more complicated literal strings that can be hard to read.

To avoid escaping characters, C++11 introduced raw string literals that do not process escape sequences. In this recipe, you will learn how to use the various forms of raw string literals.

Getting ready

In this recipe, and throughout the rest of the book, I will use the `s` suffix to define `basic_string` literals. This has been covered in the recipe *Creating cooked user-defined literals*.

How to do it…

To avoid escaping characters, define the string literals with the following:

1. `R"( literal )"` as the default form:

```
auto filename {R"(C:\Users\Marius\Documents\)"s};
auto pattern {R"((\w+)=(\d+)$)"s};

auto sqlselect {
  R"(SELECT *
  FROM Books
  WHERE Publisher='Paktpub'
  ORDER BY PubDate DESC)"s};
```

2. `R"delimiter( literal )delimiter"` where `delimiter` is any character sequence not present in the actual string when the sequence `)"` should actually be part of the string. Here is an example with `!!` as delimited:

```
auto text{ R"!!(This text contains both "( and )".)!!"s };
std::cout << text << std::endl;
```

How it works…

When string literals are used, escapes are not processed, and the actual content of the string is written between the delimiter (in other words, what you see is what you get). The following example shows what appears as the same raw literal string; however, the second one still contains escaped characters. Since these are not processed in the case of string literals, they will be printed as they are in the output:

```
auto filename1 {R"(C:\Users\Marius\Documents\)"s};
auto filename2 {R"(C:\\Users\\Marius\\Documents\\)"s};

// prints C:\Users\Marius\Documents\
std::cout << filename1 << std::endl;

// prints C:\\Users\\Marius\\Documents\\
std::cout << filename2 << std::endl;
```

In case the text has to contain the `)"` sequence, then a different delimiter must be used, in the `R"delimiter( literal )delimiter"` form. According to the standard, the possible characters in a delimiter can be as follows:

any member of the basic source character set except: space, the left parenthesis (the right parenthesis), the backslash \, and the control characters representing horizontal tab, vertical tab, form feed, and newline.

Raw string literals can be prefixed by one of `L`, `u8`, `u`, and `U` to indicate a wide, UTF-8, UTF-16, or UTF-32 string literal. The following are examples of such string literals. Note that the presence of string literal `operator ""s` at the end of the string makes the compiler deduce the type as various string classes and not character arrays:

```
auto t1{ LR"(text)"  };  // const wchar_t*
auto t2{ u8R"(text)" };  // const char*
auto t3{ uR"(text)"  };  // const char16_t*
auto t4{ UR"(text)"  };  // const char32_t*

auto t5{ LR"(text)"s  }; // wstring
auto t6{ u8R"(text)"s }; // string
auto t7{ uR"(text)"s  }; // u16string
auto t8{ UR"(text)"s  }; // u32string
```

See also

- *Creating cooked user-defined literals*

Creating a library of string helpers

The string types from the standard library are a general purpose implementation that lacks many helpful methods, such as changing the case, trimming, splitting, and others that may address different developer needs. Third-party libraries that provide rich sets of string functionalities exist. However, in this recipe, we will look at implementing several simple, yet helpful, methods you may often need in practice. The purpose is rather to see how string methods and standard general algorithms can be used for manipulating strings, but also to have a reference to reusable code that can be used in your applications.

In this recipe, we will implement a small library of string utilities that will provide functions for the following:

- Changing a string to lowercase or uppercase.
- Reversing a string.
- Trimming white spaces from the beginning and/or the end of the string.
- Trimming a specific set of characters from the beginning and/or the end of the string.
- Removing occurrences of a character anywhere in the string.
- Tokenizing a string using a specific delimiter.

Getting ready

The string library we will be implementing should work with all the standard string types, `std::string`, `std::wstring`, `std::u16string`, and `std::u32string`. To avoid specifying long names such as `std::basic_string<CharT, std::char_traits<CharT>, std::allocator<CharT>>`, we will use the following alias templates for strings and string streams:

```
template <typename CharT>
using tstring =
   std::basic_string<CharT, std::char_traits<CharT>,
                     std::allocator<CharT>>;

template <typename CharT>
using tstringstream =
   std::basic_stringstream<CharT, std::char_traits<CharT>,
                           std::allocator<CharT>>;
```

To implement these string helper functions, we need to include the header `<string>` for strings and `<algorithm>` for the general standard algorithms we will use.

In all the examples in this recipe, we will use the standard user-defined literal operators for strings from C++14, for which we need to explicitly use the `std::string_literals` namespace.

How to do it…

1. To convert a string to lowercase or uppercase, apply the `tolower()` or `toupper()` functions on the characters of a string using the general purpose algorithm `std::transform()`:

```
template<typename CharT>
inline tstring<CharT> to_upper(tstring<CharT> text)
{
  std::transform(std::begin(text), std::end(text),
                 std::begin(text), toupper);
  return text;
}

template<typename CharT>
inline tstring<CharT> to_lower(tstring<CharT> text)
{
  std::transform(std::begin(text), std::end(text),
                 std::begin(text), tolower);
  return text;
}
```

2. To reverse a string, use the general purpose algorithm `std::reverse()`:

```
template<typename CharT>
inline tstring<CharT> reverse(tstring<CharT> text)
{
  std::reverse(std::begin(text), std::end(text));
  return text;
}
```

3. To trim a string, at the beginning, end, or both, use `std::basic_string`'s methods `find_first_not_of()` and `find_last_not_of()`:

```
template<typename CharT>
inline tstring<CharT> trim(tstring<CharT> const & text)
{
  auto first{ text.find_first_not_of(' ') };
  auto last{ text.find_last_not_of(' ') };
  return text.substr(first, (last - first + 1));
}

template<typename CharT>
inline tstring<CharT> trimleft(tstring<CharT> const & text)
{
  auto first{ text.find_first_not_of(' ') };
  return text.substr(first, text.size() - first);
}

template<typename CharT>
inline tstring<CharT> trimright(tstring<CharT> const & text)
{
  auto last{ text.find_last_not_of(' ') };
  return text.substr(0, last + 1);
}
```

4. To trim characters in a given set from a string, use overloads of `std::basic_string`'s methods `find_first_not_of()` and `find_last_not_of()`, that take a string parameter that defines the set of characters to look for:

```
template<typename CharT>
inline tstring<CharT> trim(tstring<CharT> const & text,
                           tstring<CharT> const & chars)
{
  auto first{ text.find_first_not_of(chars) };
```

```
  auto last{ text.find_last_not_of(chars) };
  return text.substr(first, (last - first + 1));
}

template<typename CharT>
inline tstring<CharT> trimleft(tstring<CharT> const & text,
                               tstring<CharT> const & chars)
{
  auto first{ text.find_first_not_of(chars) };
  return text.substr(first, text.size() - first);
}

template<typename CharT>
inline tstring<CharT> trimright(tstring<CharT> const &text,
                                tstring<CharT> const &chars)
{
  auto last{ text.find_last_not_of(chars) };
  return text.substr(0, last + 1);
}
```

5. To remove characters from a string, use `std::remove_if()` and `std::basic_string::erase()`:

```
template<typename CharT>
inline tstring<CharT> remove(tstring<CharT> text,
                             CharT const ch)
{
  auto start = std::remove_if(
                 std::begin(text), std::end(text),
                 [=](CharT const c) {return c ==  ch; });
  text.erase(start, std::end(text));
  return text;
}
```

6. To split a string based on a specified delimiter, use `std::getline()` to read from an `std::basic_stringstream` initialized with the content of the string. The tokens extracted from the stream are pushed into a vector of strings:

```
template<typename CharT>
inline std::vector<tstring<CharT>> split
   (tstring<CharT> text, CharT const delimiter)
{
  auto sstr = tstringstream<CharT>{ text };
  auto tokens = std::vector<tstring<CharT>>{};
  auto token = tstring<CharT>{};
  while (std::getline(sstr, token, delimiter))
  {
    if (!token.empty()) tokens.push_back(token);
  }
  return tokens;
}
```

How it works…

For implementing the utility functions from the library, we have two options:

- Functions would modify a string passed by a reference.
- Functions would not alter the original string but return a new string.

The second option has the advantage that it preserves the original string, which may be helpful in many cases. Otherwise, in those cases, you would first have to make a copy of the string and alter the copy. The implementation provided in this recipe takes the second approach.

The first functions we implemented in the *How to do it…* section were `to_upper()` and `to_lower()`. These functions change the content of a string either to uppercase or lowercase. The simplest way to implement this is using the `std::transform()` standard algorithm. This is a general purpose algorithm that applies a function to every element of a range (defined by a begin and end iterator) and stores the result in another range for which only the begin iterator needs to be specified. The output range can be the same as the input range, which is exactly what we did to transform the string. The applied function is `toupper()` or `tolower()`:

```
auto ut{ string_library::to_upper("this is not UPPERCASE"s) };
// ut = "THIS IS NOT UPPERCASE"

auto lt{ string_library::to_lower("THIS IS NOT lowercase"s) };
// lt = "this is not lowercase"
```

The next function we considered was `reverse()`, that, as the name implies, reverses the content of a string. For this, we used the `std::reverse()` standard algorithm. This general purpose algorithm reverses the elements of a range defined by a begin and end iterator:

```
auto rt{string_library::reverse("cookbook"s)}; // rt = "koobkooc"
```

When it comes to trimming, a string can be trimmed at the beginning, end, or both sides. Because of that, we implemented three different functions: `trim()` for trimming at both ends, `trimleft()` for trimming at the beginning of a string, and `trimright()` for trimming at the end of a string. The first version of the functions trims only spaces. In order to find the right part to trim, we use the `find_first_not_of()` and `find_last_not_of()` methods of `std::basic_string`. These return the first and last characters in the string that are not the specified character. Subsequently, a call to the `substr()` method of `std::basic_string` returns a new string. The `substr()` method takes an index in the string and a number of elements to copy to the new string:

```
auto text1{"   this is an example   "s};
// t1 = "this is an example"
auto t1{ string_library::trim(text1) };
// t2 = "this is an example   "
auto t2{ string_library::trimleft(text1) };
// t3 = "   this is an example"
auto t3{ string_library::trimright(text1) };
```

It could be sometimes useful to trim other characters and then spaces from a string. In order to do that, we provided overloads for the trimming functions that specify a set of

characters to be removed. That set is also specified as a string. The implementation is very similar to the previous one because both `find_first_not_of()` and `find_last_not_of()` have overloads that take a string containing the characters to be excluded from the search:

```
auto chars1{" !%\n\r"s};
auto text3{"!!  this % needs a lot\rof trimming  !\n"s};
auto t7{ string_library::trim(text3, chars1) };
// t7 = "this % needs a lot\rof trimming"
auto t8{ string_library::trimleft(text3, chars1) };
// t8 = "this % needs a lot\rof trimming  !\n"
auto t9{ string_library::trimright(text3, chars1) };
// t9 = "!!  this % needs a lot\rof trimming"
```

If removing characters from any part of the string is necessary, the trimming methods are not helpful because they only treat a contiguous sequence of characters at the start and end of a string. For that, however, we implemented a simple `remove()` method. This uses the `std:remove_if()` standard algorithm. Both `std::remove()` and `std::remove_if()` work in a way that may not be very intuitive at first. They remove elements that satisfy the criteria from a range defined by a first and last iterator by rearranging the content of the range (using move assignment). The elements that need to be removed are placed at the end of the range, and the function returns an iterator to the first element in the range that represents the removed elements. This iterator basically defines the new end of the range that was modified. If no element was removed, the returned iterator is the end iterator of the original range. The value of this returned iterator is then used to call the `std::basic_string::erase()` method that actually erases the content of the string defined by two iterators. The two iterators in our case are the iterator returned by `std::remove_if()` and the end of the string:

```
auto text4{"must remove all * from text**"s};
auto t10{ string_library::remove(text4, '*') };
// t10 = "must remove all  from text"
auto t11{ string_library::remove(text4, '!') };
// t11 = "must remove all * from text**"
```

The last method we implemented splits the content of a string based on a specified delimiter. There are various ways to implement this. In this implementation, we used `std::getline()`. This function reads characters from an input stream until a specified delimiter is found and places the characters in a string. Before starting to read from the input buffer, it calls `erase()` on the output string to clear its content. Calling this method in a loop produces tokens that are placed in a vector. In our implementation, empty tokens were skipped from the result set:

```
auto text5{"this text will be split   "s};
auto tokens1{ string_library::split(text5, ' ') };
// tokens1 = {"this", "text", "will", "be", "split"}
auto tokens2{ string_library::split(""s, ' ') };
// tokens2 = {}
```

See also

- *Creating cooked user-defined literals*
- *Creating type aliases and alias templates* recipe of Chapter 1, *Learning Modern Core Language Features*

Verifying the format of a string using regular expressions

Regular expressions are a language intended for performing pattern matching and replacements in texts. C++11 provides support for regular expressions within the standard library through a set of classes, algorithms, and iterators available in the header `<regex>`. In this recipe, we will see how regular expressions can be used to verify that a string matches a pattern (examples can include verifying an e-mail or IP address formats).

Getting ready

Throughout this recipe, we will explain whenever necessary the details of the regular expressions that we use. However, you should have at least some basic knowledge of regular expressions in order to use the C++ standard library for regular expressions. A description of regular expressions syntax and standards is beyond the purpose of this book; if you are not familiar with regular expressions, it is recommended that you read more about them before continuing with the recipes that focus on regular expressions.

How to do it…

In order to verify that a string matches a regular expression, perform the following steps:

1. Include headers `<regex>` and `<string>` and the namespace `std::string_literals` for C++14 standard user-defined literals for strings:

```
#include <regex>
#include <string>
using namespace std::string_literals;
```

2. Use raw string literals to specify the regular expression to avoid escaping backslashes (that can occur frequently). The following regular expression validates most e-mails formats:

```
auto pattern {R"(^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$)"s};
```

3. Create an `std::regex`/`std::wregex` object (depending on the character set that is used) to encapsulate the regular expression:

```
auto rx = std::regex{pattern};
```

4. To ignore casing or specify other parsing options, use an overloaded constructor that has an extra parameter for regular expression flags:

```
auto rx = std::regex{pattern, std::regex_constants::icase};
```

5. Use `std::regex_match()` to match the regular expression to an entire string:

```
auto valid = std::regex_match("marius@domain.com"s, rx);
```

How it works…

Considering the problem of verifying the format of e-mail addresses, even though this may look like a trivial problem, in practice it is hard to find a simple regular expression that covers all the possible cases for valid e-mail formats. In this recipe, we will not try to find that ultimate regular expression, but rather to apply a regular expression that is good enough for most cases. The regular expression we will use for this purpose is this:

```
^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$
```

The following table explains the structure of the regular expression:

Part	Description
`^`	Start of string
`[A-Z0-9._%+-]+`	At least one character in the range A-Z, 0-9, or one of -, %, + or - that represents the local part of the email address
`@`	Character @
`[A-Z0-9.-]+`	At least one character in the range A-Z, 0-9, or one of -, %, + or - that represents the hostname of the domain part
`\.`	A dot that separates the domain hostname and label
`[A-Z]{2,}`	The DNS label of a domain that can have between 2 and 63 characters
`$`	End of the string

Bear in mind that in practice a domain name is composed of a hostname followed by a dot-separated list of DNS labels. Examples include `localhost`, `gmail.com`, or `yahoo.co.uk`. This regular expression we are using does not match domains without DNS labels, such as localhost (an e-mail, such as `root@localhost` is a valid e-mail). The domain name can also be an IP address specified in brackets, such as `[192.168.100.11]` (as in `john.doe@[192.168.100.11]`). E-mail addresses containing such domains will not match the regular expression defined above. Even though these rather rare formats will not be matched, the regular expression can cover most of the e-mail formats.

The regular expression in the example in this chapter is provided for didactical purposes only, and it is not intended for being used as it is in production code. As explained earlier, this sample does not cover all possible e-mail formats.

We began by including the necessary headers, `<regex>` for regular expressions and `<string>` for strings. The `is_valid_email()` function shown in the following (that basically contains the samples from the *How to do it…* section) takes a string representing an e-mail address and returns a boolean indicating whether the e-mail has a valid format or not. We first construct an `std::regex` object to encapsulate the regular expression indicated with the raw string literal. Using raw string literals is helpful because it avoids escaping backslashes

that are used for escape characters in regular expressions too. The function then calls `std::regex_match()`, passing the input text and the regular expression:

```
bool is_valid_email_format(std::string const & email)
{
  auto pattern {R"(^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$)"s};

  auto rx = std::regex{pattern};

  return std::regex_match(email, rx);
}
```

The `std::regex_match()` method tries to match the regular expression against the entire string. If successful it returns `true`, otherwise `false`:

```
auto ltest = [](std::string const & email)
{
  std::cout << std::setw(30) << std::left
            << email << " : "
            << (is_valid_email_format(email) ?
               "valid format" : "invalid format")
            << std::endl;
};

ltest("JOHN.DOE@DOMAIN.COM"s);         // valid format
ltest("JOHNDOE@DOMAIL.CO.UK"s);        // valid format
ltest("JOHNDOE@DOMAIL.INFO"s);         // valid format
ltest("J.O.H.N_D.O.E@DOMAIN.INFO"s);   // valid format
ltest("ROOT@LOCALHOST"s);              // invalid format
ltest("john.doe@domain.com"s);         // invalid format
```

In this simple test, the only e-mails that do not match the regular expression are `ROOT@LOCALHOST` and `john.doe@domain.com`. The first contains a domain name without a dot-prefixed DNS label and that case is not covered in the regular expression. The second contains only lowercase letters, and in the regular expression, the valid set of characters for both the local part and the domain name was uppercase letters, A to Z.

Instead of complicating the regular expression with additional valid characters (such as `[A-Za-z0-9._%+-]`), we can specify that the match can ignore the case. This can be done with an additional parameter to the constructor of the `std::basic_regex` class. The available constants for this purpose are defined in the `regex_constants` namespace. The following slight change to the `is_valid_email_format()` will make it ignore the case and allow e-mails with both lowercase and uppercase letters to correctly match the regular expression:

```
bool is_valid_email_format(std::string const & email)
{
  auto rx = std::regex{
    R"(^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$)"s,
    std::regex_constants::icase};

  return std::regex_match(email, rx);
}
```

This `is_valid_email_format()` function is pretty simple, and if the regular expression was provided as a parameter along with the text to match, it could be used for matching anything. However, it would be nice to be able to handle with a single function not only multi-byte strings (`std::string`) but also wide strings (`std::wstring`). This can be achieved by creating a function template where the character type is provided as a template parameter:

```
template <typename CharT>
using tstring = std::basic_string<CharT, std::char_traits<CharT>,
                                  std::allocator<CharT>>;
```

```
template <typename CharT>
bool is_valid_format(tstring<CharT> const & pattern,
                     tstring<CharT> const & text)
{
  auto rx = std::basic_regex<CharT>{
    pattern, std::regex_constants::icase };

  return std::regex_match(text, rx);
}
```

We start by creating an alias template for `std::basic_string` in order to simplify its use. The new `is_valid_format()` function is a function template very similar to our implementation of `is_valid_email()`. However, we now use `std::basic_regex<CharT>` instead of the typedef `std::regex`, which is `std::basic_regex<char>`, and the pattern is provided as the first argument. We now implement a new function called `is_valid_email_format_w()` for wide strings that relies on this function template. The function template, however, can be reused for implementing other validations, such as if a license plate has a particular format:

```
bool is_valid_email_format_w(std::wstring const & text)
{
  return is_valid_format(
    LR"(^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$)"s,
    text);
}

auto ltest2 = [](auto const & email)
{
  std::wcout << std::setw(30) << std::left
     << email << L" : "
     << (is_valid_email_format_w(email) ? L"valid" : L"invalid")
     << std::endl;
};

ltest2(L"JOHN.DOE@DOMAIN.COM"s);       // valid
ltest2(L"JOHNDOE@DOMAIL.CO.UK"s);      // valid
ltest2(L"JOHNDOE@DOMAIL.INFO"s);       // valid
ltest2(L"J.O.H.N_D.O.E@DOMAIN.INFO"s); // valid
ltest2(L"ROOT@LOCALHOST"s);            // invalid
ltest2(L"john.doe@domain.com"s);       // valid
```

Of all the examples shown above, the only one that does not match is `ROOT@LOCAHOST`, as already expected.

The `std::regex_match()` method has, in fact, several overloads, and some of them have a parameter that is a reference to an `std::match_results` object to store the result of the match. If there is no match, then `std::match_results` is empty and its size is 0. Otherwise, if there is a match, the `std::match_results` object is not empty and its size is 1 plus the number of matched subexpressions.

The following version of the function uses the mentioned overloads and returns the matched subexpressions in an `std::smatch` object. Note that the regular expression is changed, as three caption groups are defined— one for the local part, one for the hostname part of the domain, and one for the DNS label. If the match is successful, then the `std::smatch` object will contain four submatch objects: the first to match the entire string, the second for the first capture group (the local part), the third for the second capture group (the hostname), and the fourth for the third and last capture group (the DNS label). The result is returned in a tuple, where the first item actually indicates success or failure:

```
std::tuple<bool, std::string, std::string, std::string>
is_valid_email_format_with_result(std::string const & email)
{
```

```
  auto rx = std::regex{
    R"(^([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,})$)"s,
    std::regex_constants::icase };
  auto result = std::smatch{};
  auto success = std::regex_match(email, result, rx);

  return std::make_tuple(
    success,
    success ? result[1].str() : ""s,
    success ? result[2].str() : ""s,
    success ? result[3].str() : ""s);
}
```

Following the preceding code, we use C++17 structured bindings to unpack the content of the tuple into named variables:

```
auto ltest3 = [](std::string const & email)
{
  auto [valid, localpart, hostname, dnslabel] =
   is_valid_email_format_with_result(email);

  std::cout << std::setw(30) << std::left
     << email << " : "
     << std::setw(10) << (valid ? "valid" : "invalid")
     << "local=" << localpart
     << ";domain=" << hostname
     << ";dns=" << dnslabel
     << std::endl;
};

ltest3("JOHN.DOE@DOMAIN.COM"s);
ltest3("JOHNDOE@DOMAIL.CO.UK"s);
ltest3("JOHNDOE@DOMAIL.INFO"s);
ltest3("J.O.H.N_D.O.E@DOMAIN.INFO"s);
ltest3("ROOT@LOCALHOST"s);
ltest3("john.doe@domain.com"s);
```

The output of the program will be as follows:

```
JOHN.DOE@DOMAIN.COM            : valid
   local=JOHN.DOE;domain=DOMAIN;dns=COM
JOHNDOE@DOMAIL.CO.UK           : valid
   local=JOHNDOE;domain=DOMAIL.CO;dns=UK
JOHNDOE@DOMAIL.INFO            : valid
   local=JOHNDOE;domain=DOMAIL;dns=INFO
J.O.H.N_D.O.E@DOMAIN.INFO      : valid
   local=J.O.H.N_D.O.E;domain=DOMAIN;dns=INFO
ROOT@LOCALHOST                 : invalid
   local=;domain=;dns=
john.doe@domain.com            : valid
   local=john.doe;domain=domain;dns=com
```

There’s more…

There are multiple versions of regular expressions, and the C++ standard library supports six of them: ECMAScript, basic POSIX, extended POSIX, awk, grep, and egrep (grep with option `-E`). The default grammar used is ECMAScript, and in order to use another, you explicitly have to specify the grammar when defining the regular expression. In addition to specifying the grammar, you can also specify parsing options, such as matching by ignoring the case.

The standard library provides more classes and algorithms than what we have seen so far. The main classes available in the library are the following (all of them are class templates and, for convenience, `typedef`s are provided for different character types):

- The class template `std::basic_regex` defines the regular expression object:

```
typedef basic_regex<char>    regex;
typedef basic_regex<wchar_t> wregex;
```

- The class template `std::sub_match` represents a sequence of characters that matches a capture group; this class is actually derived from `std::pair`, and its `first` and `second` members represent iterators to the first and the one-past-end characters in the match sequence; if there is no match sequence, the two iterators are equal:

```
typedef sub_match<const char *>            csub_match;
typedef sub_match<const wchar_t *>         wcsub_match;
typedef sub_match<string::const_iterator>  ssub_match;
typedef sub_match<wstring::const_iterator> wssub_match;
```

- The class template `std::match_results` is a collection of matches; the first element is always a full match in the target, and the other elements are matches of subexpressions:

```
typedef match_results<const char *>            cmatch;
typedef match_results<const wchar_t *>         wcmatch;
typedef match_results<string::const_iterator>  smatch;
typedef match_results<wstring::const_iterator> wsmatch;
```

The algorithms available in the regular expressions standard library are the following:

- `std::regex_match()`: This tries to match a regular expression (represented by a `std::basic_regex` instance) to an entire string.
- `std::regex_search()`: This tries to match a regular expression (represented by a `std::basic_regex` instance) to a part of a string (including the entire string).
- `std::regex_replace()`: This replaces matches from a regular expression according to a specified format.

The iterators available in the regular expressions standard library are the following:

- `std::regex_interator`: A constant forward iterator used to iterate through the occurrences of a pattern in a string. It has a pointer to an `std::basic_regex` that must live until the

iterator is destroyed. Upon creation and when incremented, the iterator calls `std::regex_search()` and stores a copy of the `std::match_results` object returned by the algorithm.

- `std::regex_token_iterator`: A constant forward iterator used to iterate through the submatches of every match of a regular expression in a string. Internally, it uses an `std::regex_iterator` to step through the submatches. Since it stores a pointer to an `std::basic_regex` instance, the regular expression object must live until the iterator is destroyed.

See also

- *Parsing the content of a string using regular expressions*
- *Replacing the content of a string using regular expressions*
- *Using structured bindings to handle multi-return values* recipe of Chapter 1, *Learning Modern Core Language Features*

Parsing the content of a string using regular expressions

In the previous recipe, we have looked at how to use `std::regex_match()` to verify that the content of a string matches a particular format. The library provides another algorithm called `std::regex_search()` that matches a regular expression against any part of a string, and not only the entire string as `regex_match()` does. This function, however, does not allow searching through all the occurrences of a regular expression in an input string. For this purpose, we need to use one of the iterator classes available in the library.

In this recipe, you will learn how to parse the content of a string using regular expressions. For this purpose, we will consider the problem of parsing a text file containing name-value pairs. Each such pair is defined on a different line having the format `name = value`, but lines starting with a `#` represent comments and must be ignored. The following is an example:

```
#remove # to uncomment the following lines
timeout=120
server = 127.0.0.1

#retrycount=3
```

Getting ready

For general information about regular expressions support in C++11, refer to the *Verifying the format of a string using regular expressions* recipe. Basic knowledge of regular expressions is required for proceeding with this recipe.

In the following examples, `text` is a variable defined as shown here:

```
auto text {
  R"(
    #remove # to uncomment the following lines
    timeout=120
    server = 127.0.0.1

    #retrycount=3
  )"s};
```

How to do it…

In order to search for occurrences of a regular expression through a string you should perform the following:

1. Include headers `<regex>` and `<string>` and the namespace `std::string_literals` for C++14 standard user-defined literals for strings:

```
#include <regex>
#include <string>
using namespace std::string_literals;
```

2. Use raw string literals to specify the regular expression to avoid escaping backslashes (that can occur frequently). The following regular expression validates the file format proposed earlier:

```
auto pattern {R"(^(?!#)(\w+)\s*=\s*([\w\d]+[\w\d._,\-:]*)$)"s};
```

3. Create an `std::regex`/`std::wregex` object (depending on the character set that is used) to encapsulate the regular expression:

```
auto rx = std::regex{pattern};
```

4. To search for the first occurrence of a regular expression in a given text, use the general purpose algorithm `std::regex_search()` (example 1):

```
auto match = std::smatch{};
if (std::regex_search(text, match, rx))
{
  std::cout << match[1] << '=' << match[2] << std::endl;
}
```

5. To find all the occurrences of a regular expression in a given text, use the iterator `std::regex_iterator` (example 2):

```
auto end = std::sregex_iterator{};
for (auto it=std::sregex_iterator{ std::begin(text),
                                   std::end(text), rx };
     it != end; ++it)
{
  std::cout << ''' << (*it)[1] << "'='"
            << (*it)[2] << ''' << std::endl;
}
```

6. To iterate through all the subexpressions of a match, use the iterator `std::regex_token_iterator` (example 3):

```
auto end = std::sregex_token_iterator{};
for (auto it = std::sregex_token_iterator{
                  std::begin(text),  std::end(text), rx };
     it != end; ++it)
{
  std::cout << *it << std::endl;
}
```

How it works…

A simple regular expression that can parse the input file shown earlier may look like this:

```
^(?!#)(\w+)\s*=\s*([\w\d]+[\w\d._,\-:]*)$
```

This regular expression is supposed to ignore all lines that start with a `#`; for those that do not start with `#`, match a name followed by the equal sign and then a value that can be composed of alphanumeric characters and several other characters (underscore, dot, comma, and so on). The exact meaning of this regular expression is explained as follows:

Part	Description
`^`	Start of line
`(?!#)`	A negative lookahead that makes sure that it is not possible to match the # character
`(\w)+`	A capturing group representing an identifier of at least a one word character
`\s*`	Any white spaces
`=`	Equal sign
`\s*`	Any white spaces
`([\w\d]+[\w\d._,\-:]*)`	A capturing group representing a value that starts with an alphanumeric character, but can also contain a dot, comma, backslash, hyphen, colon, or an underscore.
`$`	End of line

We can use `std::regex_search()` to search for a match anywhere in the input text. This algorithm has several overloads, but in general they work in the same way. You must specify the range of characters to work through, an output `std::match_results` object that will contain the result of the match, and a `std::basic_regex` object representing the regular expression and matching flags (that define the way the search is done). The function returns `true` if a match was found or `false` otherwise.

In the first example from the previous section (see the 4th list item), `match` is an instance of `std::smatch` that is a `typedef` of `std::match_results` with `string::const_iterator` as the template type. If a match was found, this object will contain the matching information in a sequence of values for all matched subexpressions. The submatch at index 0 is always the entire match. The submatch at index 1 is the first subexpression that was matched, the submatch at index 2 is the second subexpression that was matched, and so on. Since we have two capturing groups (that are subexpressions) in our regular expression, the `std::match_results` will have three submatches in case of success. The identifier representing the name is at index 1, and the value after the equal sign is at index 2. Therefore, this code only prints the following:

```
timeout=120
```

The `std::regex_search()` algorithm is not able to iterate through all the possible matches in a text. To do that, we need to use an iterator. `std::regex_iterator` is intended for this purpose. It allows not only iterating through all the matches, but also accessing all the submatches of a match. The iterator actually calls `std::regex_search()` upon construction and on each increment, and it remembers the result `std::match_results` from the call. The default constructor creates an iterator that represents the end of the sequence and can be used to test when the loop through the matches should stop.

In the second example from the previous section (see the 5th list item), we first create an end of sequence iterator, and then we start iterating through all the possible matches. When constructed, it will call `std::regex_match()`, and if a match is found, we can access its results through the current iterator. This will continue until no match is found (end of the sequence). This code will print the following output:

```
'timeout'='120'
'server'='127.0.0.1'
```

An alternative to `std::regex_iterator` is `std::regex_token_iterator`. This works similar to the way `std::regex_iterator` works and, in fact, it contains such an iterator internally, except that it enables us to access a particular subexpression from a match. This is shown in the third example in the *How to do it...* section (the the 6th list item). We start by creating an end-of-sequence iterator and then loop through the matches until the end-of-sequence is reached. In the constructor we used, we did not specify the index of the subexpression to access through the iterator; therefore, the default value of 0 is used. That means this program will print the entire matches:

```
timeout=120
server = 127.0.0.1
```

If we wanted to access only the first subexpression (that means the names in our case), all we had to do was specify the index of the subexpression in the constructor of the token iterator. This time, the output that we get is only the names:

```
auto end = std::sregex_token_iterator{};
for (auto it = std::sregex_token_iterator{ std::begin(text),
                std::end(text), rx, 1 };
     it != end; ++it)
{
  std::cout << *it << std::endl;
}
```

An interesting thing about the token iterator is that it can return the unmatched parts of the string if the index of the subexpressions is `-1`, in which case it returns an `std::match_results` object that corresponds to the sequence of characters between the last match and the end of the sequence:

```
auto end = std::sregex_token_iterator{};
for (auto it = std::sregex_token_iterator{ std::begin(text),
                std::end(text), rx, -1 };
     it != end; ++it)
{
  std::cout << *it << std::endl;
}
```

This program will output the following (note that the empty lines are actually part of the output):

```
#remove # to uncomment the following lines



#retrycount=3
```

See also

- *Verifying the format of a string using regular expressions*
- *Replacing the content of a string using regular expressions*

Replacing the content of a string using regular expressions

In the last two recipes, we have looked at how to match a regular expression on a string or a part of a string and iterate through matches and submatches. The regular expression library also supports text replacement based on regular expressions. In this recipe, we will see how to use `std::regex_replace()` to perform such text transformations.

Getting ready

For general information about regular expressions support in C++11, refer to the *Verifying the format of a string using regular expressions* recipe.

How to do it…

In order to perform text transformations using regular expressions, you should perform the following:

1. Include the `<regex>` and `<string>` and the namespace `std::string_literals` for C++14 standard user defined literals for strings:

```
#include <regex>
#include <string>
using namespace std::string_literals;
```

2. Use the `std::regex_replace()` algorithm with a replacement string as the third argument. Consider this example: replace all words composed of exactly three characters that are either `a`, `b`, or `c` with three hyphens:

```
auto text{"abc aa bca ca bbbb"s};
auto rx = std::regex{ R"(\b[a|b|c]{3}\b)"s };
auto newtext = std::regex_replace(text, rx, "---"s);
```

3. Use the `std::regex_replace()` algorithm with match identifiers prefixed with a `$` for the third argument. For example, replace names in the “lastname, firstname” with names in the format “firstname lastname”, as follows:

```
auto text{ "bancila, marius"s };
auto rx = std::regex{ R"((\w+),\s*(\w+))"s };
auto newtext = std::regex_replace(text, rx, "$2 $1"s);
```

How it works…

The `std::regex_replace()` algorithm has several overloads with different types of parameters, but the meaning of the parameters is as follows:

- The input string on which the replacement is performed.
- An `std::basic_regex` object that encapsulates the regular expression used to identify the parts of the strings to be replaced.
- The string format used for replacement.
- Optional matching flags.

The return value is, depending on the overload used, either a string or a copy of the output iterator provided as an argument. The string format used for replacement can either be a simple string or a match identifier indicated with a `$` prefix:

- `$&` indicates the entire match.
- `$1`, `$2`, `$3`, and so on, indicate the first, second, third submatch, and so on.
- `$`` indicates the part of the string before the first match.
- `$'` indicates the part of the string after the last match.

In the first example shown in the *How to do it…* section, the initial text contains two words made of exactly three `a`, `b`, or `c` characters, `abc` and `bca`. The regular expression indicates an expression of exactly three characters between word boundaries. That means a subtext, such as `bbbb`, will not match the expression. The result of the replacement is that the string text will be `--- aa --- ca bbbb`.

Additional flags for the match can be specified to the `std::regex_replace()` algorithm. By default, the matching flag is `std::regex_constants::match_default` that basically specifies ECMAScript as the grammar used for constructing the regular expression. If we want, for instance, to replace only the first occurrence, then we can specify `std::regex_constants::format_first_only`. In the next example, the result is `--- aa bca ca bbbb` as the replacement stops after the first match is found:

```
auto text{ "abc aa bca ca bbbb"s };
auto rx = std::regex{ R"(\b[a|b|c]{3}\b)"s };
auto newtext = std::regex_replace(text, rx, "---"s,
                 std::regex_constants::format_first_only);
```

The replacement string, however, can contain special indicators for the whole match, a particular submatch, or the parts that were not matched, as explained earlier. In the second example shown in the *How to do it…* section, the regular expression identifies a word of at least one character, followed by a coma and possible white spaces and then another word of at least one character. The first word is supposed to be the last name and the second word is supposed to be the first name. The replacement string has the `$2 $1` format. This is an instruction to replace the matched expression (in this example, the entire original string) with another string formed of the second submatch followed by space and then the first submatch.

In this case, the entire string was a match. In the next example, there will be multiple matches inside the string, and they will all be replaced with the indicated string. In this example, we are replacing the indefinite article *a* when preceding a word that starts with a vowel (this, of course, does not cover words that start with a vowel sound) with the indefinite article *an*:

```
auto text{"this is a example with a error"s};
auto rx = std::regex{R"(\ba ((a|e|i|u|o)\w+))"s};
auto newtext = std::regex_replace(text, rx, "an $1");
```

The regular expression identifies the letter *a* as a single word (`\b` indicates a word boundary, so `\ba` means a word with a single letter *a*) followed by a space and a word of at least two characters starting with a vowel. When such a match is identified, it is replaced with a string formed of the fixed string *an* followed by a space and the first subexpression of the match, which is the word itself. In this example, the `newtext` string will be *this is an example with an error*.

Apart from the identifiers of the subexpressions (`$1`, `$2`, and so on), there are other identifiers for the entire match (`$&`), the part of the string before the first match (`$`` `) and the part of the string after the last match (`$'`). In the last example, we change the format of a date from `dd.mm.yyyy` to `yyyy.mm.dd`, but also show the matched parts:

```
auto text{"today is 1.06.2016!!"s};
auto rx =
   std::regex{R"((\d{1,2})(\.|-|/)(\d{1,2})(\.|-|/)(\d{4}))"s};
// today is 2016.06.1!!
auto newtext1 = std::regex_replace(text, rx, R"($5$4$3$2$1)");
// today is [today is ][1.06.2016][!!]!!
auto newtext2 = std::regex_replace(text, rx, R"([$`][$&][$'])");
```

The regular expression matches a one- or two-digit number followed by a dot, hyphen, or slash; followed by another one- or two-digit number; then a dot, hyphen, or slash; and last a four-digit number.

For `newtext1`, the replacement string is `$5$4$3$2$1`; that means year, followed by the second separator, then month, the first separator, and finally day. Therefore, for the input string *"today is 1.06.2016!"*, the result is *"today is 2016.06.1!!"*.

For `newtext2`, the replacement string is `[$`][$&][$']`; that means the part before the first match, followed by the entire match, and finally the part after the last match are in square brackets. However, the result is not *"[!!][1.06.2016][today is]"* as you perhaps might expect at a first glance, but *"today is [today is][1.06.2016][!!]!!"*. The reason is that what is replaced is the matched expression, and, in this case, that is only the date (*"1.06.2016"*). This substring is replaced with another string formed of the all parts of the initial string.

See also

- *Verifying the format of a string using regular expressions*
- *Parsing the content of a string using regular expressions*

Using string_view instead of constant string references

When working with strings, temporary objects are created all the time, even if you might not be really aware of it. Many times the temporary objects are irrelevant and only serve the purpose of copying data from one place to another (for example, from a function to its caller). This represents a performance issue because they require memory allocation and data copying, which is desirable to be avoided. For this purpose, the C++17 standard provides a new string class template called `std::basic_string_view` that represents a non-owning constant reference to a string (that is, a sequence of characters). In this recipe, you will learn when and how you should use this class.

Getting ready

The `string_view` class is available in the namespace `std` in the `string_view` header.

How to do it…

You should use `std::string_view` to pass a parameter to a function (or return a value from a function), instead of `std::string const &` unless your code needs to call other functions that take `std::string` parameters (in which case, conversions would be necessary):

```
    std::string_view get_filename(std::string_view str)
    {
      auto const pos1 {str.find_last_of('')};
      auto const pos2 {str.find_last_of('.')};
      return str.substr(pos1 + 1, pos2 - pos1 - 1);
    }

    char const file1[] {R"(c:\test\example1.doc)"};
    auto name1 = get_filename(file1);

    std::string file2 {R"(c:\test\example2)"};
    auto name2 = get_filename(file2);

    auto name3 = get_filename(std::string_view{file1, 16});
```

How it works…

Before we look at how the new string type works, let's consider the following example of a function that is supposed to extract the name of a file without its extension. This is basically how you would write the function from the previous section before C++17.

Note that in this example the file separator is \ (backslash) as in Windows. For Linux-based systems, it has to be changed to / (slash).

```
std::string get_filename(std::string const & str)
{
  auto const pos1 {str.find_last_of('')};
  auto const pos2 {str.find_last_of('.')};
  return str.substr(pos1 + 1, pos2 - pos1 - 1);
}

auto name1 = get_filename(R"(c:\test\example1.doc)"); // example1
auto name2 = get_filename(R"(c:\test\example2)");     // example2
if(get_filename(R"(c:\test\_sample_.tmp)").front() == '_') {}
```

This is a relatively simple function. It takes a constant reference to an `std::string` and identifies a substring bounded by the last file separator and the last dot that basically represents a filename without an extension (and without folder names).

The problem with this code, however, is that it creates one, two, or, possibly, even more temporaries, depending on the compiler optimizations. The function parameter is a constant `std::string` reference, but the function is called with a string literal, which means `std::string` needs to be constructed from the literal. These temporaries need to allocate and copy data, which is both time- and resource-consuming. In the last example, all we want to do is check whether the first character of the filename is an underscore, but we create at least two temporary string objects for that purpose.

The `std::basic_string_view` class template is intended to solve this problem. This class template is very similar to `std::basic_string`, the two having almost the same interface. The reasons for this is that the `std::basic_string_view` is intended to be used instead of a constant reference to an `std::basic_string` without further code changes. Just like with `std::basic_string`, there are specializations for all types of standard characters:

```
typedef basic_string_view<char>     string_view;
typedef basic_string_view<wchar_t>  wstring_view;
typedef basic_string_view<char16_t> u16string_view;
typedef basic_string_view<char32_t> u32string_view;
```

The `std::basic_string_view` class template defines a reference to a constant contiguous sequence of characters. As the name implies, it represents a view and cannot be used to modify the reference sequence of characters. An `std::basic_string_view` object has a relatively small size because all that it needs is a pointer to the first character in the sequence and the length. It can be constructed not only from an `std::basic_string` object but also from a pointer and a length or from a null-terminated sequence of characters (in which case, it will require an initial traversing of the string in order to find the length). Therefore, the `std::basic_string_view` class template can also be used as a common interface for multiple types of strings (as long as data only needs to be read). On the other hand,

converting from an `std::basic_string_view` to an `std::basic_string` is easy because the former has both a `to_string()` and a converting `operator std::basic_string` to create a new `std::basic_string` object.

Passing `std::basic_string_view` to functions and returning `std::basic_string_view` still creates temporaries of this type, but these are small size objects on the stack (a pointer and a size could be 16 bytes for 64-bit platforms); therefore, they should incur fewer performance costs than allocating heap space and copying data.

Note that Microsoft implementation of `std::basic_string` provides an optimization for small strings, by having a statically allocated buffer of 16 characters that does not involve heap operations, which are only required when the size of the string exceeds 16 characters.

In addition to the methods that are identical to those available in `std::basic_string`, the `std::basic_string_view` has two more:

- `remove_prefix()`: Shrinks the view by incrementing the start with *N* characters and decrementing the length with *N* characters.
- `remove_suffix()`: Shrinks the view by decrementing the length with *N* characters.

The two member functions are used in the following example to trim an `std::string_view` from spaces, both at the beginning and the end. The implementation of the function first looks for the first element that is not a space and then for the last element that is not a space. Then, it removes from the end everything after the last non-space character, and from the beginning everything until the first non-space character. The function returns the new view trimmed at both ends:

```
std::string_view trim_view(std::string_view str)
{
  auto const pos1{ str.find_first_not_of(" ") };
  auto const pos2{ str.find_last_not_of(" ") };
  str.remove_suffix(str.length() - pos2 - 1);
  str.remove_prefix(pos1);

  return str;
}

auto sv1{ trim_view("sample") };
auto sv2{ trim_view("  sample") };
auto sv3{ trim_view("sample  ") };
auto sv4{ trim_view("  sample  ") };

auto s1{ sv1.to_string() };
auto s2{ sv2.to_string() };
auto s3{ sv3.to_string() };
auto s4{ sv4.to_string() };
```

When using an `std::basic_string_view`, you must be aware of two things: you cannot change the underlying data referred by a view and you must manage the lifetime of the data, as the view is a non-owning reference.

See also

- *Creating a library of string helpers*

Exploring Functions

The recipes included in this chapter are as follows:

- Defaulted and deleted functions
- Using lambdas with standard algorithms
- Using generic lambdas
- Writing a recursive lambda
- Writing a function template with a variable number of arguments
- Using fold expressions to simplify variadic function templates
- Implementing higher-order functions map and fold
- Composing functions into a higher-order function
- Uniformly invoking anything callable

Introduction

Functions are a fundamental concept in programming; regardless of the topic we discuss, we end up writing functions. Trying to cover functions in a single chapter is not only hard but also not very rational. This book contains recipes related to functions in all the other chapters. This chapter, however, covers modern language features related to functions and callable objects, with a focus on lambda expressions, concepts from functional languages such as higher-order functions, and type-safe functions with a variable number of arguments.

Defaulted and deleted functions

In C++, classes have special members (constructors, destructors, and operators) that may be either implemented by default by the compiler or supplied by the developer. However, the rules for what can be default implemented are a bit complicated and can lead to problems. On the other hand, developers sometimes want to prevent objects from being copied, moved, or constructed in a particular way. That is possible by implementing different tricks using these special members. The C++11 standard has simplified many of these by allowing functions to be deleted or defaulted in the manner we will see in the next section.

Getting started

For this recipe, you need to know what special member functions are and what copyable and moveable means.

How to do it…

Use the following syntax to specify how functions should be handled:

- To default a function, use `=default` instead of the function body. Only special class member functions that have defaults can be defaulted:

```
struct foo
{
  foo() = default;
};
```

- To delete a function, use `=delete` instead of the function body. Any function, including non-member functions, can be deleted:

```
struct foo
{
  foo(foo const &) = delete;
};

void func(int) = delete;
```

Use defaulted and deleted functions to achieve various design goals, such as the following examples:

- To implement a class that is not copyable, and implicitly not movable, declare the copy operations as deleted:

```
class foo_not_copyable
{
public:
  foo_not_copyable() = default;

  foo_not_copyable(foo_not_copyable const &) = delete;
  foo_not_copyable& operator=(foo_not_copyable const&) = delete;
};
```

- To implement a class that is not copyable, but is movable, declare the copy operations as deleted and explicitly implement the move operations (and provide any additional constructors that are needed):

```
class data_wrapper
{
  Data* data;
public:
  data_wrapper(Data* d = nullptr) : data(d) {}
  ~data_wrapper() { delete data; }

  data_wrapper(data_wrapper const&) = delete;
  data_wrapper& operator=(data_wrapper const &) = delete;

  data_wrapper(data_wrapper&& o) :data(std::move(o.data))
  {
    o.data = nullptr;
  }

  data_wrapper& operator=(data_wrapper&& o)
  {
    if (this != &o)
    {
      delete data;
```

```
            data = std::move(o.data);
            o.data = nullptr;
        }

        return *this;
    }
};
```

- To ensure a function is called only with objects of a specific type, and perhaps prevent type promotion, provide deleted overloads for the function (the following example with free functions can also be applied to any class member functions):

```
template <typename T>
void run(T val) = delete;

void run(long val) {} // can only be called with long integers
```

How it works…

A class has several special members that can be implemented, by default, by the compiler. These are the default constructor, copy constructor, move constructor, copy assignment, move assignment, and destructor (for a discussion on move semantics, refer to the *Implementing move semantics* recipe from Chapter 9, *Robustness and Performance*). If you don't implement them, then the compiler does it so that instances of a class can be created, moved, copied, and destructed. However, if you explicitly provide one or more of these special methods, then the compiler will not generate the others according to the following rules:

- If a user-defined constructor exists, the default constructor is not generated by default.
- If a user-defined virtual destructor exists, the default constructor is not generated by default.
- If a user-defined move constructor or move assignment operator exists, then the copy constructor and copy assignment operator are not generated by default.
- If a user-defined copy constructor, move constructor, copy assignment operator, move assignment operator, or destructor exists, then the move constructor and move assignment operator are not generated by default.
- If a user-defined copy constructor or destructor exists, then the copy assignment operator is generated by default.
- If a user-defined copy assignment operator or destructor exists, then the copy constructor is generated by default.

Note that the last two rules in the preceding list are deprecated rules and may no longer be supported by your compiler.

Sometimes, developers need to provide empty implementations of these special members or hide them in order to prevent the instances of the class from being constructed in a specific manner. A typical example is a class that is not supposed to be copyable. The classical pattern for this is to provide a default constructor and hide the copy constructor and copy assignment operators. While this works, the explicitly defined default constructor ensures the class is no longer considered trivial and, therefore, a POD type (that can be constructed with `reinterpret_cast`). The modern alternative to this is using a deleted function as shown in the preceding section.

When the compiler encounters `=default` in the definition of a function, it will provide the default implementation. The rules for special member functions mentioned earlier still apply. Functions can be declared `=default` outside the body of a class if and only if they are inlined:

```
class foo
{
public:
  foo() = default;
```

```
  inline foo& operator=(foo const &);
};

inline foo& foo::operator=(foo const &) = default;
```

When the compiler encounters the `=delete` in the definition of a function, it will prevent the calling of the function. However, the function is still considered during overload resolution, and only if the deleted function is the best match, the compiler generates an error. For example, by giving the previously defined overloads for the `run()` function, only calls with long integers are possible. Calls with arguments of any other type, including `int`, for which an automatic type promotion to `long` exists, will determine a deleted overload to be considered the best match and therefore the compiler will generate an error:

```
run(42);  // error, matches a deleted overload
run(42L); // OK, long integer arguments are allowed
```

Note that previously declared functions cannot be deleted, as the `=delete` definition must be the first declaration in a translation unit:

```
void forward_declared_function();
// ...
void forward_declared_function() = delete; // error
```

The rule of thumb (also known as The Rule of Five) for class special member functions is that, if you explicitly define any copy constructor, move constructor, copy assignment operator, move assignment operator, or destructor, then you must either explicitly define or default all of them.

Using lambdas with standard algorithms

One of the most important modern features of C++ is lambda expressions, also referred to as lambda functions or simply lambdas. Lambda expressions enable us to define anonymous function objects that can capture variables in the scope and be invoked or passed as arguments to functions. Lambdas are useful for many purposes, and in this recipe, we will see how to use them with standard algorithms.

Getting ready

In this recipe, we discuss standard algorithms that take an argument that is a function or predicate applied to the elements it iterates through. You need to know what unary and binary functions are and what predicates and comparison functions are. You also need to be familiar with function objects because lambda expressions are syntactic sugar for function objects.

How to do it…

You should prefer to use lambda expressions to pass callbacks to standard algorithms instead of functions or function objects:

- Define anonymous lambda expressions in the place of the call if you only need to use the lambda in a single place:

```
auto numbers =
  std::vector<int>{ 0, 2, -3, 5, -1, 6, 8, -4, 9 };
auto positives = std::count_if(
  std::begin(numbers), std::end(numbers),
  [](int const n) {return n > 0; });
```

- Define a named lambda, that is, one assigned to a variable (usually with the `auto` specifier for the type), if you need to call the lambda in multiple places:

```
auto ispositive = [](int const n) {return n > 0; };
auto positives = std::count_if(
  std::begin(numbers), std::end(numbers), ispositive);
```

- Use generic lambda expressions if you need lambdas that only differ in their argument types (available since C++14):

```
auto positives = std::count_if(
  std::begin(numbers), std::end(numbers),
  [](auto const n) {return n > 0; });
```

How it works…

The non-generic lambda expression shown on the second bullet earlier takes a constant integer and returns `true` if it is greater than `0`, or `false` otherwise. The compiler defines an unnamed function object with the call operator having the signature of the lambda expression:

```
struct __lambda_name__
{
  bool operator()(int const n) const { return n > 0; }
};
```

The way the unnamed function object is defined by the compiler depends on the way we define the lambda expression that can capture variables, use the `mutable` specifier or exception specifications, or have a trailing return type. The `__lambda_name__` function object shown earlier is actually a simplification of what the compiler generates because it also defines a default copy and move constructor, a default destructor, and a deleted assignment operator.

It must be well understood that the lambda expression is actually a class. In order to call it, the compiler needs to instantiate an object of the class. The object instantiated from a lambda expression is called a lambda closure.

In the next example, we want to count the number of elements in a range that are greater than or equal to 5 and less than or equal to 10. The lambda expression, in this case, will look like this:

```
auto numbers = std::vector<int>{ 0, 2, -3, 5, -1, 6, 8, -4, 9 };
auto start{ 5 };
auto end{ 10 };
auto inrange = std::count_if(
         std::begin(numbers), std::end(numbers),
         [start, end](int const n) {
            return start <= n && n <= end;});
```

This lambda captures two variables, `start` and `end`, by copy (that is, value). The resulting unnamed function object created by the compiler looks very much like the one we defined earlier. With the default and deleted special members mentioned earlier, the class looks like this:

```
class __lambda_name_2__
{
  int start_;
  int end_;
public:
  explicit __lambda_name_2__(int const start, int const end) :
    start_(start), end_(end)
  {}

  __lambda_name_2__(const __lambda_name_2__&) = default;
  __lambda_name_2__(__lambda_name_2__&&) = default;
  __lambda_name_2__& operator=(const __lambda_name_2__&)
     = delete;
  ~__lambda_name_2__() = default;

  bool operator() (int const n) const
  {
    return start_ <= n && n <= end_;
  }
};
```

The lambda expression can capture variables by copy (or value) or by reference, and different combinations of the two are possible. However, it is not possible to capture a variable multiple times, and it is only possible to have `&` or `=` at the beginning of the capture list.

A lambda can only capture variables from an enclosing function scope. It cannot capture variables with static storage duration (that is, variables declared in a namespace scope or with the `static` or `external` specifier).

The following table shows various combinations for lambda captures semantics.

Lambda	Description
`[](){}`	Does not capture anything
`[&](){}`	Captures everything by reference
`[=](){}`	Captures everything by copy
`[&x](){}`	Capture only `x` by reference
`[x](){}`	Capture only `x` by copy
`[&x...](){}`	Capture pack extension `x` by reference
`[x...](){}`	Capture pack extension `x` by copy
`[&, x](){}`	Captures everything by reference except for `x` that is captured by copy
`[=, &x](){}`	Captures everything by copy except for `x` that is captured by reference
`[&, this](){}`	Captures everything by reference except for pointer `this` that is captured by copy (`this` is always captured by copy)
`[x, x](){}`	Error, `x` is captured twice
`[&, &x](){}`	Error, everything is captured by reference, cannot specify again to capture `x` by reference
`[=, =x](){}`	Error, everything is captured by copy, cannot specify again to capture `x` by copy
`[&this](){}`	Error, pointer `this` is always captured by copy
`[&, =](){}`	Error, cannot capture everything both by copy and by reference

The general form of a lambda expression, as of C++17, looks like this:

```
[capture-list](params) mutable constexpr exception attr -> ret
{ body }
```

All parts shown in this syntax are actually optional except for the capture list, that can, however, be empty, and the body, that can also be empty. The parameter list can actually be omitted if no parameters are needed. The return type does not need to be specified, as the compiler can infer it from the type of the returned expression. The `mutable` specifier (that tells the compiler the lambda can actually modify variables captured by copy), the `constexpr` specifier (that tells the compiler to generate a `constexpr` call operator), and the

exception specifiers and attributes are all optional.

The simplest possible lambda expression is `[]{}`, though it is often written as `[](){}`.

There’s more…

There are cases where lambda expressions only differ in the type of their arguments. In this case, the lambdas can be written in a generic way, just like templates, but using the `auto` specifier for the type parameters (no template syntax is involved). This is addressed in the next recipe, mentioned in the *See also* section.

See also

- *Using generic lambdas*
- *Writing a recursive lambda*

Using generic lambdas

In the preceding recipe, we saw how to write lambda expressions and use them with standard algorithms. In C++, lambdas are basically syntactic sugar for unnamed function objects, which are classes that implement the call operator. However, just like any other function, this can be implemented generically with templates. C++14 takes advantage of this and introduces generic lambdas that do not need to specify actual types for their parameters and use the `auto` specifier instead. Though not referred with this name, generic lambdas are basically lambda templates. They are useful in cases where we want to use the same lambda but with different types of parameter.

Getting started

It is recommended that you read the preceding recipe, *Using lambdas with standard algorithms*, before you continue with this one.

How to do it…

Write generic lambdas:

- By using the `auto` specifier instead of actual types for lambda expression parameters.
- When you need to use multiple lambdas that only differ by their parameter types.

The following example shows a generic lambda used with the `std::accumulate()` algorithm first with a vector of integers and then with a vector of strings.

```
auto numbers =
  std::vector<int>{0, 2, -3, 5, -1, 6, 8, -4, 9};
auto texts =
  std::vector<std::string>{"hello"s, " "s, "world"s, "!"s};

auto lsum = [](auto const s, auto const n) {return s + n;};

auto sum = std::accumulate(
  std::begin(numbers), std::end(numbers), 0, lsum);
  // sum = 22

auto text = std::accumulate(
  std::begin(texts), std::end(texts), ""s, lsum);
  // sum = "hello world!"s
```

How it works…

In the example from the previous section, we have defined a named lambda expression, that is, a lambda expression that has its closure assigned to a variable. This variable is then passed as an argument to the `std::accumulate()` function. This general algorithm takes the begin and the end iterators that define a range, an initial value to accumulate over, and a function that is supposed to accumulate each value in the range to the total. This function takes a first parameter representing the currently accumulated value and a second parameter representing the current value to accumulate to the total, and it returns the new accumulated value. Note that I did not use the term `add` because this can be used for other things than just adding. It can also be used for calculating a product, concatenating, or other operations that aggregate values together.

The two calls to `std::accumulate()` in this example are almost the same, only the types of the arguments are different:

- In the first call, we pass iterators to a range of integers (from a `vector<int>`), 0 for the initial sum and a lambda that adds two integers and returns their sum. This produces a sum of all integers in the range; for this example, it is 22.
- In the second call, we pass iterators to a range of strings (from a `vector<string>`), an empty string for the initial value, and a lambda that concatenates two strings by adding them together and returning the result. This produces a string that contains all the strings in the range put together one after an other; for this example, the result is *“hello world!”*.

Though generic lambdas can be defined anonymously in the place where they are called, it does not really make sense because the very purpose of a generic lambda (that is basically, as mentioned earlier, a lambda expression template) is to be reused, as shown in the example from the *How to do it…* section.

When defining this lambda expression used with multiple calls to `std::accumulate()`, instead of specifying concrete types for the lambda parameters (such as `int` or `std::string`) we used the `auto` specifier and let the compiler deduce the type. When encountering a lambda expression that has the `auto` specifier for a parameter type, the compiler generates an unnamed function object that has a call operator template. For the generic lambda expression in this example, the function object would look like this:

```
struct __lambda_name__
{
  template<typename T1, typename T2>
  auto operator()(T1 const s, T2 const n) const { return s + n; }

   __lambda_name__(const __lambda_name__&) = default;
   __lambda_name__(__lambda_name__&&) = default;
   __lambda_name__& operator=(const __lambda_name__&) = delete;
   ~__lambda_name__() = default;
};
```

The call operator is a template with a type parameter for each parameter in the lambda that was specified with `auto`. The return type of the call operator is also `auto`, which means the

compiler will deduce it from the type of the returned value. This operator template will be instantiated with the actual types the compiler will identify in the context where the generic lambda is used.

See also

- *Using lambdas with standard algorithms*
- *Using auto whenever possible* recipe of Chapter 1, *Learning Modern Core Language Features*

Writing a recursive lambda

Lambdas are basically unnamed function objects, which means that it should be possible to call them recursively. Indeed, they can be called recursively; however, the mechanism for doing it is not obvious, as it requires assigning the lambda to a function wrapper and capturing the wrapper by reference. Though it can be argued that a recursive lambda does not really make sense and a function is probably a better design choice, in this recipe we will look at how to write a recursive lambda.

Getting ready

To demonstrate how to write a recursive lambda, we will consider the well-known example of the Fibonacci function. This is usually implemented recursively in C++, as follows:

```
constexpr int fib(int const n)
{
  return n <= 2 ? 1 : fib(n - 1) + fib(n - 2);
}
```

How to do it…

In order to write a recursive lambda function, you must perform the following:

- Define the lambda in a function scope.
- Assign the lambda to an `std::function` wrapper.
- Capture the `std::function` object by reference in the lambda in order to call it recursively.

The following are examples of recursive lambdas:

- A recursive Fibonacci lambda expression in the scope of a function that is invoked from the scope where it is defined:

```
void sample()
{
  std::function<int(int const)> lfib =
    [&lfib](int const n)
    {
      return n <= 2 ? 1 : lfib(n - 1) + lfib(n - 2);
    };

  auto f10 = lfib(10);
}
```

- A recursive Fibonacci lambda expression returned by a function, that can be invoked from any scope:

```
std::function<int(int const)> fib_create()
{
  std::function<int(int const)> f = [](int const n)
  {
    std::function<int(int const)> lfib = [&lfib](int n)
    {
      return n <= 2 ? 1 : lfib(n - 1) + lfib(n - 2);
    };
    return lfib(n);
  };
  return f;
}

void sample()
{
  auto lfib = fib_create();
  auto f10 = lfib(10);
}
```

How it works…

The first thing you need to consider when writing a recursive lambda is that a lambda expression is a function object and, in order to call it recursively from the lambda's body, the lambda must capture its closure (that is, the instantiation of the lambda). In other words, the lambda must capture itself and this has several implications:

- First of all, the lambda must have a name; an unnamed lambda cannot be captured in order to be called again.
- Secondly, the lambda can only be defined in a function scope. The reason for this is that a lambda can only capture variables from a function scope; it cannot capture any variable that has a static storage duration. Objects defined in a namespace scope or with the static or external specifiers have static storage duration. If the lambda was defined in a namespace scope, its closure would have static storage duration and therefore the lambda would not capture it.
- The third implication is that the type of the lambda closure cannot remain unspecified, that is, be declared with the auto specifier. It is not possible for a variable declared with the auto type specifier to appear in its own initializer because the type of the variable is not known when the initializer is being processed. Therefore, you must specify the type of the lambda closure. The way we can do this is using the general purpose function wrapper `std::function`.
- Last, but not least, the lambda closure must be captured by reference. If we capture by copy (or value), then a copy of the function wrapper is made, but the wrapper is uninitialized when the capturing is done. We end up with an object that we are not able to call. Even though the compiler will not complain about capturing by value, when the closure is invoked, an `std::bad_function_call` is thrown.

In the first example from the *How to do it…* section, the recursive lambda is defined inside another function called `sample()`. The signature and the body of the lambda expression are the same as those of the regular recursive function `fib()` defined in the introductory section. The lambda closure is assigned to a function wrapper called `lfib` that is then captured by reference by the lambda and called recursively from its body. Since the closure is captured by reference, it will be initialized at the time it has to be called from the lambda's body.

In the second example, we have defined a function that returns the closure of a lambda expression that, in turn, defines and invokes a recursive lambda with the argument it was, in turn, invoked with. This is a pattern that must be implemented when a recursive lambda needs to be returned from a function. This is necessary because the lambda closure must still be available at the time the recursive lambda is called. If it is destroyed before that, we are left with a dangling reference and calling it will cause the program to terminate abnormally. This erroneous situation is exemplified in the following sample:

```
// this implementation of fib_create is faulty
std::function<int(int const)> fib_create()
{
```

```
    std::function<int(int const)> lfib = [&lfib](int const n)
    {
      return n <= 2 ? 1 : lfib(n - 1) + lfib(n - 2);
    };

    return lfib;
  }

  void sample()
  {
    auto lfib = fib_create();
    auto f10 = lfib(10);       // crash
  }
```

The solution for this is to create two nested lambda expressions as shown in the *How to do it…* section. The `fib_create()` method returns a function wrapper that when invoked creates the recursive lambda that captures itself. This is slightly and subtly, yet fundamentally, different from the implementation shown in the preceding sample. The outer `f` lambda does not capture anything, especially by reference; therefore, we don't have the issue with dangling references. However, when invoked, it creates a closure of the nested lambda, the actual lambda we are interested in calling and returns the result of applying that recursive `lfib` lambda to its parameter.

Writing a function template with a variable number of arguments

It is sometimes useful to write functions with a variable number of arguments or classes with a variable number of members. Typical examples include functions such as `printf` that take a format and a variable number of arguments, or classes such as `tuple`. Before C++11, the former was possible only with the use of variadic macros (that enable writing only type-unsafe functions) and the latter was not possible at all. C++11 introduced variadic templates, which are templates with a variable number of arguments that make it possible to write both type-safe function templates with a variable number of arguments and also class templates with a variable number of members. In this recipe, we will look at writing function templates.

Getting ready

Functions with a variable number of arguments are called *variadic functions*. Function templates with a variable number of arguments are called *variadic function templates*. Knowledge of C++ variadic macros (`va_start`, `va_end`, `va_arg` and `va_copy`, `va_list`) is not necessary for learning how to write variadic function templates, but it represents a good starting point.

We have already used variadic templates in our previous recipes, but this one will provide detailed explanations.

How to do it…

In order to write variadic function templates, you must perform the following steps:

1. Define an overload with a fixed number of arguments to end compile-time recursion if the semantics of the variadic function template require it (refer to `[1]` in the following code).
2. Define a template parameter pack to introduce a template parameter that can hold any number of arguments, including zero; these arguments can be either types, non-types, or templates (refer to `[2]`).
3. Define a function parameter pack to hold any number of function arguments, including zero; the size of the template parameter pack and the corresponding function parameter pack is the same and can be determined with the `sizeof...` operator (refer to `[3]`).
4. Expand the parameter pack in order to replace it with the actual arguments being supplied (refer to `[4]`).

The following example that illustrates all the preceding points, is a variadic function template that adds a variable number of arguments using `operator+`:

```
template <typename T>                // [1] overload with fixed
T add(T value)                       //     number of arguments
{
  return value;
}

template <typename T, typename... Ts> // [2] typename... Ts
T add(T head, Ts... rest)             // [3] Ts... rest
{
  return head + add(rest...);         // [4] rest...
}
```

How it works…

At a first look, the preceding implementation looks like recursion, because function `add()` calls itself, and in a way it is, but it is a compile-time recursion that does not incur any sort of runtime recursion and overhead. The compiler actually generates several functions with a different number of arguments, based on the variadic function template usage, so it is only function overloading that is involved and not any sort of recursion. However, implementation is done as if parameters would be processed in a recursive manner with an end condition.

In the preceding code we can identify the following key parts:

- `Typename... Ts` is a template parameter pack that indicates a variable number of template type arguments.
- `Ts... rest` is a function parameter pack that indicates a variable number of function arguments.
- `Rest...` is an expansion of the function parameter pack.

The position of the ellipsis is not syntactically relevant. `typename... Ts`, `typename ... Ts`, *and* `typename ...Ts` *are all equivalent.*

In the `add(T head, Ts... rest)` parameter, `head` is the first element of the list of arguments, and `...rest` is a pack with the rest of the parameters in the list (this can be zero or more). In the body of the function, `rest...` is an expansion of the function parameter pack. This means the compiler replaces the parameter pack with its elements in their order. In the `add()` function, we basically add the first argument to the sum of the remaining arguments, which gives the impression of a recursive processing. This recursion ends when there is a single argument left, in which case the first `add()` overload (with a single argument) is called and returns the value of its argument.

This implementation of the function template `add()` enables us to write code, as shown here:

```
auto s1 = add(1, 2, 3, 4, 5);
// s1 = 15
auto s2 = add("hello"s, " "s, "world"s, "!"s);
// s2 = "hello world!"
```

When the compiler encounters `add(1, 2, 3, 4, 5)`, it generates the following functions (`arg1`, `arg2`, and so on, are not the actual names the compiler generates) that show this is actually only calls to overloaded functions and not recursion:

```
int add(int head, int arg1, int arg2, int arg3, int arg4)
{return head + add(arg1, arg2, arg3, arg4);}
int add(int head, int arg1, int arg2, int arg3)
{return head + add(arg1, arg2, arg3);}
int add(int head, int arg1, int arg2)
{return head + add(arg1, arg2);}
int add(int head, int arg1)
{return head + add(arg1);}
int add(int value)
```

```
    {return value;}
```

With GCC and Clang, you can use the `__PRETTY_FUNCTION__` macro to print the name and the signature of the function.

By adding a `std::cout << __PRETTY_FUNCTION__ << std::endl` at the beginning of the two functions we wrote, we get the following when running the code:

```
    T add(T, Ts ...) [with T = int; Ts = {int, int, int, int}]
    T add(T, Ts ...) [with T = int; Ts = {int, int, int}]
    T add(T, Ts ...) [with T = int; Ts = {int, int}]
    T add(T, Ts ...) [with T = int; Ts = {int}]
    T add(T) [with T = int]
```

Since this is a function template, it can be used with any type that supports `operator+`. The other example, `add("hello"s, " "s, "world"s, "!"s)`, produces the *"hello world!"* string. However, the `std::basic_string` type has different overloads for `operator+`, including one that can concatenate a string to a character, so we should be able to also write the following:

```
    auto s3 = add("hello"s, ' ', "world"s, '!');
    // s3 = "hello world!"
```

However, that will generate compiler errors as follows (note that I actually replaced `std::basic_string<char, std::char_traits<char>, std::allocator<char> >` with string *"hello world"* for simplicity):

```
In instantiation of 'T add(T, Ts ...) [with T = char; Ts = {string, char}]':
16:29:   required from 'T add(T, Ts ...) [with T = string; Ts = {char, string, char}]'
22:46:   required from here
16:29: error: cannot convert 'string' to 'char' in return
 In function 'T add(T, Ts ...) [with T = char; Ts = {string, char}]':
17:1: warning: control reaches end of non-void function [-Wreturn-type]
```

What happens is that the compiler generates the code shown next where the return type is the same as the type of the first argument. However, the first argument is either a `std::string` or a `char` (again, `std::basic_string<char, std::char_traits<char>, std::allocator<char> >` was replaced with `string` for simplicity). In cases where `char` is the type of the first argument, the type of the return value `head+add(...)` that is an `std::string` does not match the function return type and does not have an implicit conversion to it:

```
    string add(string head, char arg1, string arg2, char arg3)
    {return head + add(arg1, arg2, arg3);}
    char add(char head, string arg1, char arg2)
    {return head + add(arg1, arg2);}
    string add(string head, char arg1)
    {return head + add(arg1);}
    char add(char value)
    {return value;}
```

We can fix this by modifying the variadic function template to have `auto` for the return type instead of `T`. In this case, the return type is always inferred from the return expression, and in our example, it will be `std::string` in all cases:

```
    template <typename T, typename... Ts>
    auto add(T head, Ts... rest)
    {
      return head + add(rest...);
    }
```

It should be further added that a parameter pack can appear in a brace-initialization and its size can be determined using the `sizeof...` operator. Also, variadic function templates do

not necessarily imply compile-time recursion as we have shown in this recipe. All these are shown in the following example where we define a function that creates a tuple with an even number of members. We first use `sizeof...(a)` to make sure that we have an even number of arguments and assert by generating a compiler error otherwise. The `sizeof...` operator can be used with both template parameter packs and function parameter packs. `sizeof...(a)` and `sizeof...(T)` would produce the same value. Then, we create and return a tuple. The template parameter pack `T` is expanded (with `T...`) into the type arguments of the `std::tuple` class template, and the function parameter pack `a` is expanded (with `a...`) into the values for the tuple members using brace initialization:

```
template<typename... T>
auto make_even_tuple(T... a)
{
  static_assert(sizeof...(a) % 2 == 0,
                "expected an even number of arguments");
  std::tuple<T...> t { a... };

  return t;
}

auto t1 = make_even_tuple(1, 2, 3, 4); // OK

// error: expected an even number of arguments
auto t2 = make_even_tuple(1, 2, 3);
```

See also

- *Using fold expressions to simplify variadic function templates*
- *Creating raw user-defined literals* recipe of Chapter 2, *Working with Numbers and Strings*

Using fold expressions to simplify variadic function templates

In this chapter, we are discussing folding several times; this is an operation that applies a binary function to a range of values to produce a single value. We have seen this when we discussed variadic function templates and will see it again with higher-order functions. It turns out there is a significant number of cases where the expansion of a parameter pack in variadic function templates is basically a folding operation. To simplify writing such variadic function templates C++17 introduced fold expressions that fold an expansion of a parameter pack over a binary operator. In this recipe, we will see how to use fold expressions to simplify writing variadic function templates.

Getting ready

The examples in this recipe are based on the variadic function template `add()` that we wrote in the previous recipe, *Writing a function template with a variable number of arguments*. That implementation is a left-folding operation. For simplicity, we present the function again:

```
template <typename T>
T add(T value)
{
  return value;
}

template <typename T, typename... Ts>
T add(T head, Ts... rest)
{
  return head + add(rest...);
}
```

How to do it…

To fold a parameter pack over a binary operator, use one of the following forms:

- Left folding with a unary form `(... op pack)`:

```
template <typename... Ts>
auto add(Ts... args)
{
  return (... + args);
}
```

- Left folding with a binary form `(init op ... op pack)`:

```
template <typename... Ts>
auto add_to_one(Ts... args)
{
  return (1 + ... + args);
}
```

- Right folding with a unary form `(pack op ...)`:

```
template <typename... Ts>
auto add(Ts... args)
{
  return (args + ...);
}
```

- Right folding with a binary form `(pack op ... op init)`:

```
template <typename... Ts>
auto add_to_one(Ts... args)
{
  return (args + ... + 1);
}
```

The parentheses shown above are part of the fold expression and cannot be omitted.

How it works…

When the compiler encounters a fold expression, it expands it in one of the following expressions:

Expression	Expansion
`(... op pack)`	((pack$1 op pack$2) op …) op pack$n
`(init op ... op pack)`	(((init op pack$1) op pack$2) op …) op pack$n
`(pack op ...)`	pack$1 op (… op (pack$n-1 op pack$n))
`(pack op ... op init)`	pack$1 op (… op (pack$n-1 op (pack$n op init)))

When the binary form is used, the operator on both the left-hand and right-hand side of the ellipses must be the same, and the initialization value must not contain an unexpanded parameter pack.

The following binary operators are supported with fold expressions:

+	-	*	/	%	^	&	|	=	<	>	<<
>>	+=	-=	*=	/=	%=	^=	&=	|=	<<=	>>=	==
!=	<=	>=	&&	||	,	.*	->*.				

When using the unary form, only operators such as `*`, `+`, `&`, `|`, `&&`, `||`, and `,` (comma) are allowed with an empty parameter pack. In this case, the value of the empty pack is as follows:

`+`	`0`
`*`	`1`
`&`	`-1`
`|`	`0`
`&&`	`true`
`||`	`false`
`,`	`void()`

Now that we have the function templates implemented earlier (let's consider the left-folding version), we can write the following code:

```
auto sum = add(1, 2, 3, 4, 5);          // sum = 15
auto sum1 = add_to_one(1, 2, 3, 4, 5); // sum = 16
```

Considering the `add(1, 2, 3, 4, 5)` call, it would produce the following function:

```
int add(int arg1, int arg2, int arg3, int arg4, int arg5)
{
  return ((((arg1 + arg2) + arg3) + arg4) + arg5);
}
```

Due to the aggressive ways modern compilers do optimizations, this function can be inlined and eventually end up with an expression such as `auto sum = 1 +`

2 + 3 + 4 + 5.

There’s more…

Fold expressions work with all overloads for the supported binary operators, but do not work with arbitrary binary functions. It is possible to implement a workaround for that by providing a wrapper type to hold a value and an overloaded operator for that wrapper type:

```
template <typename T>
struct wrapper
{
  T const & value;
};

template <typename T>
constexpr auto operator<(wrapper<T> const & lhs,
                         wrapper<T> const & rhs)
{
  return wrapper<T> {
    lhs.value < rhs.value ? lhs.value : rhs.value};
}

template <typename... Ts>
constexpr auto min(Ts&&... args)
{
  return (wrapper<Ts>{args} < ...).value;
}
```

In the preceding code, `wrapper` is a simple class template that holds a constant reference to a value of type `T`. An overloaded `operator<` is provided for this class template; this overload does not return a Boolean to indicate that the first argument is less than the second, but actually an instance of the `wrapper` class type to hold the minimum value of the two arguments. The variadic function template `min()` uses this overloaded `operator<` to fold the pack of arguments expanded to instances of the wrapper class template:

```
auto m = min(1, 2, 3, 4, 5); // m = 1
```

See also

- *Implementing higher-order functions map and fold*

Implementing higher-order functions map and fold

Throughout the preceding recipes in this book, we have used the general purpose algorithms `std::transform()` and `std::accumulate()` in several examples, such as implementing string utilities to create uppercase or lowercase copies of a string or summing the values of a range. These are basically implementations of higher-order functions, `map` and `fold`. A higher-order function is a function that takes one or more other functions as arguments and applies them to a range (a list, vector, map, tree, and so on), producing either a new range or a value. In this recipe, we will see how to implement `map` and `fold` functions to work with C++ standard containers.

Getting ready

Map is a higher-order function that applies a function to the elements of a range and returns a new range in the same order.

Fold is a higher-order function that applies a combining function to the elements of the range producing a single result. Since the order of the processing can be important, there are usually two versions of this function—`foldleft`, that processes elements from left to right, and `foldright` that combines the elements from right to left.

Most descriptions of the function map indicate that it is applied to a `list`, but this is a general term that can indicate different sequential types, such as list, vector, and array, and also dictionaries (that is, maps), queues, and so on. For this reason, I prefer to use the term range when describing these higher-order functions.

How to do it…

To implement the `map` function you should:

- Use `std::transform` on containers that support iterating and assignment to the elements, such as `std::vector` or `std::list`:

```
template <typename F, typename R>
R mapf(F&& f, R r)
{
  std::transform(
    std::begin(r), std::end(r), std::begin(r),
    std::forward<F>(f));
  return r;
}
```

- Use other means such as explicit iteration and insertion for containers that do not support assignment to the elements, such as `std::map`:

```
template<typename F, typename T, typename U>
std::map<T, U> mapf(F&& f, std::map<T, U> const & m)
{
  std::map<T, U> r;
  for (auto const kvp : m)
    r.insert(f(kvp));
  return r;
}

template<typename F, typename T>
std::queue<T> mapf(F&& f, std::queue<T> q)
{
  std::queue<T> r;
  while (!q.empty())
  {
    r.push(f(q.front()));
    q.pop();
  }
  return r;
}
```

To implement the `fold` function you should:

- Use `std::accumulate()` on containers that support iterating:

```
template <typename F, typename R, typename T>
constexpr T foldl(F&& f, R&& r, T i)
{
  return std::accumulate(
    std::begin(r), std::end(r),
    std::move(i),
    std::forward<F>(f));
}

template <typename F, typename R, typename T>
constexpr T foldr(F&& f, R&& r, T i)
{
  return std::accumulate(
    std::rbegin(r), std::rend(r),
    std::move(i),
    std::forward<F>(f));
}
```

- Use other means to explicitly process containers that do not support iterating, such as

`std::queue`:

```
template <typename F, typename T>
constexpr T foldl(F&& f, std::queue<T> q, T i)
{
  while (!q.empty())
  {
    i = f(i, q.front());
    q.pop();
  }
  return i;
}
```

How it works…

In the preceding examples, we have implemented the map in a functional way, without side-effects. That means it preserves the original range and returns a new one. The arguments of the function are the function to apply and the range. In order to avoid confusion with the `std::map` container, we have called this function `mapf`. There are several overloads for `mapf` as shown earlier:

- The first overload is for containers that support iterating and assignment to its elements; this includes `std::vector`, `std::list`, and `std::array`, but also C-like arrays. The function takes an `rvalue` reference to a function and a range for which `std::begin()` and `std::end()` are defined. The range is passed by value so that modifying the local copy does not affect the original range. The range is transformed by applying the given function to each element using the standard algorithm `std::transform()`; the transformed range is then returned.
- The second overload is specialized for `std::map` that does not support direct assignment to its elements (`std::pair<T, U>`). Therefore, this overload creates a new map, then iterates through its elements using a range-based for loop, and inserts into the new map the result of applying the input function to each element of the original map.
- The third overload is specialized for `std::queue`, which is a container that does not support iterating. It can be argued that a queue is not a typical structure to map over, but for the sake of demonstrating different possible implementations, we are considering it. In order to iterate over the elements of a queue, the queue must be altered—you need to pop elements from the front until the list is empty. This is what the third overload does—it processes each element of the input queue (passed by value) and pushes the result of applying the given function to the front element of the remaining queue.

Now that we have these overloads implemented, we can apply them to a lot of containers, as shown in the following examples:

- Retain absolute values from a vector. In this example, the vector contains both negative and positive values. After applying the mapping, the result is a new vector with only positive values.

```
auto vnums =
  std::vector<int>{0, 2, -3, 5, -1, 6, 8, -4, 9};
auto r = funclib::mapf([](int const i) {
  return std::abs(i); }, vnums);
// r = {0, 2, 3, 5, 1, 6, 8, 4, 9}
```

- Square the numerical values of a list. In this example, the list contains integral values. After applying the mapping, the result is a list containing the squares of the initial values.

```
auto lnums = std::list<int>{1, 2, 3, 4, 5};
```

```
auto l = funclib::mapf([](int const i) {
  return i*i; }, lnums);
// l = {1, 4, 9, 16, 25}
```

- Rounded amounts of floating point. For this example, we need to use `std::round()`; however, this has overloads for all floating point types, which makes it impossible for the compiler to pick the right one. As a result, we either have to write a lambda that takes an argument of a specific floating point type and returns the value of `std::round()` applied to that value or create a function object template that wraps `std::round()` and enables its call operator only for floating point types. This technique is used in the following example:

```
template<class T = double>
struct fround
{
  typename std::enable_if<
    std::is_floating_point<T>::value, T>::type
  operator()(const T& value) const
  {
    return std::round(value);
  }
};

auto amounts =
  std::array<double, 5> {10.42, 2.50, 100.0, 23.75, 12.99};
auto a = funclib::mapf(fround<>(), amounts);
// a = {10.0, 3.0, 100.0, 24.0, 13.0}
```

- Uppercase the string keys of a map of words (where the key is the word and the value is the number of appearances in the text). Note that creating an uppercase copy of a string is itself a mapping operation. Therefore, in this example, we use `mapf` to apply `toupper()` to the elements of the string representing the key in order to produce an uppercase copy:

```
auto words = std::map<std::string, int>{
  {"one", 1}, {"two", 2}, {"three", 3}
};
auto m = funclib::mapf(
  [](std::pair<std::string, int> const kvp) {
    return std::make_pair(
      funclib::mapf(toupper, kvp.first),
      kvp.second);
  },
  words);
// m = {{"ONE", 1}, {"TWO", 2}, {"THREE", 3}}
```

- Normalize values from a queue of priorities—initially, the values are from 1 to 100, but we want to normalize them into two values, 1=high and 2=normal. All initial priorities that have a value up to 30 become a high priority, the others get a normal priority:

```
auto priorities = std::queue<int>();
priorities.push(10);
priorities.push(20);
priorities.push(30);
priorities.push(40);
priorities.push(50);
auto p = funclib::mapf(
  [](int const i) { return i > 30 ? 2 : 1; },
  priorities);
// p = {1, 1, 1, 2, 2}
```

To implement `fold`, we actually have to consider the two possible types of folding, that is, from left to right and from right to left. Therefore, we have provided two functions called `foldl` (for left folding) and `foldr` (for right folding). The implementations shown in the previous section are very similar—they both take a function, a range, and an initial value and call `std::algorithm()` to fold the values of the range into a single value. However, `foldl` uses direct iterators, whereas `foldr` uses reverse iterators to traverse and process the range. The second overload is a specialization for type `std::queue`, which does not have iterators.

Based on these implementations for folding, we can do the following examples:

- Adding the values of a vector of integers. In this case, both left and right folding will produce the same result. In the following examples, we pass either a lambda that takes a sum and a number and returns a new sum or the function object `std::plus<>` from the standard library that applies `operator+` to two operands of the same type (basically similar to the closure of the lambda):

```
auto vnums =
   std::vector<int>{0, 2, -3, 5, -1, 6, 8, -4, 9};

auto s1 = funclib::foldl(
   [](const int s, const int n) {return s + n; },
   vnums, 0);                // s1 = 22

auto s2 = funclib::foldl(
   std::plus<>(), vnums, 0); // s2 = 22

auto s3 = funclib::foldr(
   [](const int s, const int n) {return s + n; },
   vnums, 0);                // s3 = 22

auto s4 = funclib::foldr(
   std::plus<>(), vnums, 0); // s4 = 22
```

- Concatenating strings from a vector into a single string:

```
auto texts =
   std::vector<std::string>{"hello"s, " "s, "world"s, "!"s};

auto txt1 = funclib::foldl(
   [](std::string const & s, std::string const & n) {
   return s + n;},
   texts, ""s);    // txt1 = "hello world!"

auto txt2 = funclib::foldr(
   [](std::string const & s, std::string const & n) {
   return s + n; },
   texts, ""s);    // txt2 = "!world hello"
```

- Concatenating an array of characters into a string:

```
char chars[] = {'c','i','v','i','c'};

auto str1 = funclib::foldl(std::plus<>(), chars, ""s);
// str1 = "civic"

auto str2 = funclib::foldr(std::plus<>(), chars, ""s);
// str2 = "civic"
```

- Counting the number of words from a text based on their already computed appearances available in a `map<string, int>`:

```
auto words = std::map<std::string, int>{
   {"one", 1}, {"two", 2}, {"three", 3} };

auto count = funclib::foldl(
   [](int const s, std::pair<std::string, int> const kvp) {
      return s + kvp.second; },
   words, 0); // count = 6
```

There’s more…

These functions can be pipelined, that is, they can call one function with the result of another. The following example maps a range of integers into a range of positive integers by applying the `std::abs()` function to its elements. The result is then mapped into another range of squares. These are then summed together by applying a left fold on the range:

```
auto vnums = std::vector<int>{ 0, 2, -3, 5, -1, 6, 8, -4, 9 };

auto s = funclib::foldl(
  std::plus<>(),
  funclib::mapf(
    [](int const i) {return i*i; },
    funclib::mapf(
      [](int const i) {return std::abs(i); },
      vnums)),
  0); // s = 236
```

As an exercise, we could implement the fold function as a variadic function template, in the manner seen in a previous recipe. The function that performs the actual folding is provided as an argument:

```
template <typename F, typename T1, typename T2>
auto foldl(F&&f, T1 arg1, T2 arg2)
{
  return f(arg1, arg2);
}

template <typename F, typename T, typename... Ts>
auto foldl(F&& f, T head, Ts... rest)
{
  return f(head, foldl(std::forward<F>(f), rest...));
}
```

When we compare this with the `add()` function template that we wrote in the recipe *Writing a function template with a variable number of arguments*, we can notice several differences:

- The first argument is a function, which is perfectly forwarded when calling `foldl` recursively.
- The end case is a function that requires two arguments because the function we use for folding is a binary one (taking two arguments).
- The return type of the two functions we wrote is declared as `auto` because it must match the return type of the supplied binary function `f` that is not known until we call `foldl`:

```
  auto s1 = foldl(std::plus<>(), 1, 2, 3, 4, 5);
  // s1 = 15
  auto s2 = foldl(std::plus<>(), "hello"s, ' ', "world"s, '!');
  // s2 = "hello world!"
  auto s3 = foldl(std::plus<>(), 1); // error, too few arguments
```

See also

- *Creating a library of string helpers* recipe of Chapter 2, *Working with Numbers and Strings*
- *Writing a function template with a variable number of arguments*
- *Composing functions into a higher-order function*

Composing functions into a higher-order function

In the previous recipe, we implemented two higher-order functions, map and fold, and saw various examples of using them. At the end of the recipe, we saw how they can be pipelined to produce a final value after several transformations of the original data. Pipelining is a form of composition, which means creating one new function from two or more given functions. In the mentioned example, we didn’t actually compose functions; we only called a function with the result produced by another, but in this recipe, we will see how to actually compose functions together into a new function. For simplicity, we will only consider unary functions (functions that take only one argument).

Getting ready

Before you go forward, it is recommended that you read the previous recipe, *Implementing higher-order functions map and fol*d. It is not mandatory for understanding this recipe, but we will refer to the map and fold functions implemented here.

How to do it…

To compose unary functions into a higher-order function, you should:

- For composing two functions, provide a function that takes two functions, `f` and `g`, as arguments and returns a new function (a lambda) that returns `f(g(x))` where `x` is the argument of the composed function:

```
template <typename F, typename G>
auto compose(F&& f, G&& g)
{
  return [=](auto x) { return f(g(x)); };
}

auto v = compose(
  [](int const n) {return std::to_string(n); },
  [](int const n) {return n * n; })(-3); // v = "9"
```

- For composing a variable number of functions, provide a variadic template overload of the function described previously:

```
template <typename F, typename... R>
auto compose(F&& f, R&&... r)
{
  return [=](auto x) { return f(compose(r...)(x)); };
}

auto n = compose(
  [](int const n) {return std::to_string(n); },
  [](int const n) {return n * n; },
  [](int const n) {return n + n; },
  [](int const n) {return std::abs(n); })(-3); // n = "36"
```

How it works…

Composing two unary functions into a new one is relatively trivial. Create a template function that we called `compose()` in the earlier examples, with two arguments—`f` and `g`—that represent functions, and return a function that takes one argument `x` and returns `f(g(x))`. It is important though that the type of the value returned by the `g` function is the same as the type of the argument of the `f` function. The returned value of the compose function is a closure, that is, an instantiation of a lambda.

In practice, it is useful to be able to combine more than just two functions together. This can be achieved by writing a variadic template version of the `compose()` function. Variadic templates are explained in more detail in the *Writing a function template with a variable number of arguments* recipe. Variadic templates imply compile-time recursion by expanding the parameter pack. This implementation is very similar to the first version of `compose()`, except as follows:

- It takes a variable number of functions as arguments.
- The returned closure calls `compose()` recursively with the expanded parameter pack; recursion ends when only two functions are left, in which case, the previously implemented overload is called.

Even if the code looks like recursion is happening, this is not true recursion. It could be called compile-time recursion, but with every expansion, we get a call to another method with the same name but a different number of arguments, which does not represent recursion.

Now that we have these variadic template overloads implemented, we can rewrite the last example from the previous recipe, *Implementing higher-order functions map and fold*. Having an initial vector of integers, we map it to a new vector with only positive values by applying `std::abs()` on each element. The result is then mapped to a new vector by doubling the value of each element. Finally, the values in the resulting vector are folded together by adding them to the initial value 0:

```
auto s = compose(
  [](std::vector<int> const & v) {
    return foldl(std::plus<>(), v, 0); },
  [](std::vector<int> const & v) {
    return mapf([](int const i) {return i + i; }, v); },
  [](std::vector<int> const & v) {
    return mapf([](int const i) {return std::abs(i); }, v); })(vnums);
```

There’s more…

Composition is usually represented by a dot (`.`) or asterisk (`*`), such as `f . g` or `f * g`. We can actually do something similar in C++ by overloading `operator*` (it would make little sense to try to overload operator dot). Similar to the `compose()` function, `operator*` should work with any number of arguments; therefore, we will have two overloads, just like in the case of `compose()`:

- The first overload takes two arguments and calls `compose()` to return a new function.
- The second overload is a variadic template function that again calls `operator*` by expanding the parameter pack:

```
template <typename F, typename G>
auto operator*(F&& f, G&& g)
{
  return compose(std::forward<F>(f), std::forward<G>(g));
}

template <typename F, typename... R>
auto operator*(F&& f, R&&... r)
{
  return operator*(std::forward<F>(f), r...);
}
```

We can now simplify the actual composition of functions by applying `operator*` instead of the more verbose call to compose:

```
auto n =
  ([](int const n) {return std::to_string(n); } *
   [](int const n) {return n * n; } *
   [](int const n) {return n + n; } *
   [](int const n) {return std::abs(n); })(-3); // n = "36"

auto c =
  [](std::vector<int> const & v) {
    return foldl(std::plus<>(), v, 0); } *
  [](std::vector<int> const & v) {
    return mapf([](int const i) {return i + i; }, v); } *
  [](std::vector<int> const & v) {
    return mapf([](int const i) {return std::abs(i); }, v); };

auto s = c(vnums); // s = 76
```

See also

- *Writing a function template with a variable number of arguments*

Uniformly invoking anything callable

Developers, and especially those who implement libraries, sometimes need to invoke a callable object in a uniform manner. This can be a function, a pointer to a function, a pointer to a member function, or a function object. Examples of such cases include `std::bind`, `std::function`, `std::mem_fn`, and `std::thread::thread`. C++17 defines a standard function called `std::invoke()` that can invoke any callable object with the provided arguments. This is not intended to replace direct calls to functions or function objects, but it is useful in template metaprogramming for implementing various library functions.

Getting ready

For this recipe, you should be familiar with how to define and use function pointers.

To exemplify how `std::invoke()` can be used in different contexts, we will use the following function and class:

```
int add(int const a, int const b)
{
  return a + b;
}

struct foo
{
  int x = 0;

  void increment_by(int const n) { x += n; }
};
```

How to do it…

The `std::invoke()` function is a variadic function template that takes the callable object as the first argument and a variable list of arguments that are passed to the call. `std::invoke()` can be used to call the following:

- Free functions:

```
auto a1 = std::invoke(add, 1, 2);   // a1 = 3
```

- Free functions through pointer to function:

```
auto a2 = std::invoke(&add, 1, 2);  // a2 = 3
int(*fadd)(int const, int const) = &add;
auto a3 = std::invoke(fadd, 1, 2);  // a3 = 3
```

- Member functions through pointer to member function:

```
foo f;
std::invoke(&foo::increment_by, f, 10);
```

- Data members:

```
foo f;
auto x1 = std::invoke(&foo::x, f);  // x1 = 0
```

- Function objects:

```
foo f;
auto x3 = std::invoke(std::plus<>(),
  std::invoke(&foo::x, f), 3); // x3 = 3
```

- Lambda expressions:

```
auto l = [](auto a, auto b) {return a + b; };
auto a = std::invoke(l, 1, 2); // a = 3
```

In practice, `std:invoke()` should be used in template meta-programming for invoking a function with an arbitrary number of arguments. To exemplify such a case, we present a possible implementation for our `std::apply()` function, and also a part of the standard library as of C++17 that calls a function by unpacking the members of a tuple into the arguments of the function:

```
namespace details
{
  template <class F, class T, std::size_t... I>
  auto apply(F&& f, T&& t, std::index_sequence<I...>)
  {
    return std::invoke(
      std::forward<F>(f),
      std::get<I>(std::forward<T>(t))...);
  }
}

template <class F, class T>
auto apply(F&& f, T&& t)
{
  return details::apply(
```

```
        std::forward<F>(f),
        std::forward<T>(t),
        std::make_index_sequence<
          std::tuple_size<std::decay_t<T>>::value> {});
    }
```

How it works…

Before we see how `std::invoke()` works, let's have a short look at how different callable objects can be invoked. Given a function, obviously, the ubiquitous way of invoking it is directly passing it the necessary parameters. However, we can also invoke the function using function pointers. The trouble with function pointers is that defining the type of the pointer can be cumbersome. Using `auto` can simplify things (as shown in the following code), but in practice, you usually need to define the type of the pointer to function first and then define an object and initialize it with the correct function address. Here are several examples:

```
// direct call
auto a1 = add(1, 2);    // a1 = 3

// call through function pointer
int(*fadd)(int const, int const) = &add;
auto a2 = fadd(1, 2);   // a2 = 3

auto fadd2 = &add;
auto a3 = fadd2(1, 2);  // a3 = 3
```

Calling through a function pointer becomes more cumbersome when you need to invoke a class function through an object that is an instance of the class. The syntax for defining the pointer to a member function and invoking it is not simple:

```
foo f;
f.increment_by(3);
auto x1 = f.x;    // x1 = 3

void(foo::*finc)(int const) = &foo::increment_by;
(f.*finc)(3);
auto x2 = f.x;    // x2 = 6

auto finc2 = &foo::increment_by;
(f.*finc2)(3);
auto x3 = f.x;    // x3 = 9
```

Regardless of how cumbersome this kind of call may look, the actual problem is writing library components (functions or classes) that are able to call any of these types of callable objects, in a uniform manner. This is what benefits in practice from a standard function, such as `std::invoke()`.

The implementation details of `std::invoke()` are complex, but the way it works can be explained in simple terms. Supposing the call has the form `invoke(f, arg1, arg2, ..., argN)`, then consider the following:

- If `f` is a pointer to a member function of a `T` class, then the call is equivalent with either:
  - `(arg1.*f)(arg2, ..., argN)`, if `arg1` is an instance of `T`
  - `(arg1.get().*f)(arg2, ..., argN)`, if `arg1` is a specialization of `reference_wrapper`
  - `((*arg1).*f)(arg2, ..., argN)`, if it is otherwise
- If `f` is a pointer to a data member of a `T` class and there is a single argument, in other words, the call has the form `invoke(f, arg1)`, then the call is equivalent to either:
  - `arg1.*f` if `arg1` is an instance class `T`

    - `arg1.get().*f` if `arg1` is a specialization of `reference_wrapper`
    - `(*arg1).*f`, if it is otherwise
- If `f` is a function object, then the call is equivalent to `f(arg1, arg2, ..., argN)`

See also

- *Writing a function template with a variable number of arguments*

Preprocessor and Compilation

The recipes included in this chapter are as follows:

- Conditionally compiling your source code
- Using the indirection pattern for preprocessor stringification and concatenation
- Performing compile-time assertion checks with static_assert
- Conditionally compiling classes and functions with enable_if
- Selecting branches at compile time with constexpr if
- Providing metadata to the compiler with attributes

Introduction

In C++, compilation is the process by which source code is transformed into machine code and organized in object files that are then linked together to produce an executable. The compiler actually works on a single file at a time, produced by the preprocessor from a single source file and all the header files that it includes. This is, however, an oversimplification of what happens when we compile the code. This chapter addresses topics related to preprocessing and compilation, with a focus on various methods to perform conditional compilation, but also touching other modern topics such as using attributes for providing implementation-defined language extensions.

Conditionally compiling your source code

Conditional compilation is a simple mechanism that enables developers to maintain a single code base, but only consider some parts of the code for compilation to produce different executables, usually in order to run on different platforms, hardware or depend on different libraries or library versions. Common examples include using or ignoring code based on the compiler, platform (x86, x64, ARM, and so on), configuration (debug or release), or any user-defined specific conditions. In this recipe, we take a look at how conditional compilation works.

Getting ready

Conditional compilation is a technique used extensively for many purposes. In this recipe, we will look at several examples and explain how they work. The technique is not in any way limited to these examples. For the scope of this recipe, we will only consider the three major compilers, GCC, Clang, and VC++.

How to do it…

To conditionally compile portions of code, use the `#if`, `#ifdef`, and `#ifndef` directives (with the `#elif`, `#else`, and `#endif` directives). The general form for conditional compilation is as follows:

```
#if condition1
  text
#elif condition2
  text
#elif condition3
  text
#else
  text
#endif
```

To define macros for conditional compilation, you can use either of the following:

- A `#define` directive in your source code:

```
#define DEBUG_PRINTS
#define VERBOSITY_LEVEL 5
```

- Compiler command-line options that are specific to each compiler. Examples for the most widely used compilers are as follows:
  - For Visual C++, use `/Dname` or `/Dname=value` (where `/Dname` is equivalent to `/Dname=1`),

    for example, `cl /DVERBOSITY_LEVEL=5`.

  - For GCC and Clang, use `-D name` or `-D name=value` (where `-D name` is equivalent to `-D name=1`),

    for example, `gcc -D VERBOSITY_LEVEL=5`.

The following are typical examples of conditional compilation:

- Header guards to avoid duplicate definitions:

```
#if !defined(_UNIQUE_NAME_)
#define _UNIQUE_NAME_
class foo { };
#endif
```

- Compiler-specific code for cross-platform applications. The following is an example for printing a message to the console with the name of the compiler:

```
void show_compiler()
{
  #if defined _MSC_VER
    std::cout << "Visual C++" << std::endl;
  #elif defined __clang__
    std::cout << "Clang" << std::endl;
  #elif defined __GNUG__
    std::cout << "GCC" << std::endl;
  #else
    std::cout << "Unknown compiler" << std::endl;
  #endif
}
```

- Target-specific code for multiple architectures, for example, for conditionally compiling code for multiple compilers and architectures:

```
void show_architecture()
{
  #if defined _MSC_VER

    std::cout <<
    #if defined _M_X64
      "AMD64"
    #elif defined _M_IX86
      "INTEL x86"
    #elif defined _M_ARM
      "ARM"
    #else
      "unknown"
    #endif
  << std::endl;

  #elif defined __clang__ || __GNUG__

  std::cout <<
    #if defined __amd64__
      "AMD64"
    #elif defined __i386__
      "INTEL x86"
    #elif defined __arm__
      "ARM"
    #else
      "unknown"
    #endif
  << std::endl;

  #else
  #error Unknown compiler
  #endif
}
```

- Configuration-specific code, for example, for conditionally compiling code for debug and release builds:

```
void show_configuration()
{
  std::cout <<
  #ifdef _DEBUG
    "debug"
  #else
    "release"
  #endif
  << std::endl;
}
```

How it works…

When you use the preprocessor directives `#if`, `#ifndef`, `#ifdef`, `#elif`, `#else`, and `#endif`, the compiler will select at most one branch whose body will be included in the translation unit for compilation. The body of these directives can be any text, including other preprocessor directives. The following rules apply:

- `#if`, `#ifdef`, and `#ifndef` must be matched by a `#endif`.
- `#if` directive may have multiple `#elif` directives, but only one `#else`, which must also be the last one before `#endif`.
- `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, and `#endif` can be nested.
- `#if` directive requires a constant expression, whereas `#ifdef` and `#ifndef` require an identifier.
- The operator `defined` can be used for preprocessor constant expressions, but only in `#if` and `#elif` directives.
- `defined(identifier)` is considered `true` if `identifier` is defined, otherwise it is considered `false`.
- An identifier defined as an empty text is considered defined.
- `#ifdef identifier` is equivalent to `#if defined(identifier)`.
- `#ifndef identifier` is equivalent to `#if !defined(identifier)`.
- `defined(identifier)` and `defined identifier` are equivalent.

Header guards are one of the most common forms of conditional compilation. This technique is used to prevent the content of a header file from being compiled several times (although the header is still scanned every time in order to detect what should be included). Since headers are often included in multiple source files, having them compiled for every translation unit where they are included would produce multiple definitions for the same symbols, which is an error. Therefore, the code in headers is guarded for multiple compilations in the manner shown in the example from the previous section. The way this works, considering the given example, is that if macro `_UNIQUE_NAME_` is not defined, then the code after the `#if` directive, until `#endif`, is included into the translation unit and compiled. When that happens, the macro `_UNIQUE_NAME_` is defined with the `#define` directive. The next time the header is included in a translation unit, the macro `_UNIQUE_NAME_` is defined and the code in the body of the `#if` directive is not included in the translation unit and, therefore, not compiled again.

Note that the name of the macro must be unique throughout the application, otherwise only the code from the first header where the macro is used will be compiled, code from other headers using the same name will be ignored.

Another important example of conditional compilation is cross-platform code, which needs to account for different compilers and architectures, usually one among Intel x86, AMD64, or ARM. However, the compiler defines its own macros for the possible platforms. The samples from the *How to do it…* section show how to conditionally compile code for multiple compilers and architectures.

Note that in the mentioned example, we only consider a few architectures. In practice, there are multiple macros to identify the same architecture. Ensure that you read the documentation of each compiler before using these types of macros in your code.

Configuration-specific code is also handled with macros and conditional compilation. Compilers such as GCC and Clang do not define any special macros for debug configurations (when the `-g` flag is used). Visual C++ does define `_DEBUG` for a debug configuration, which was shown in the last example from the *How to do it…* section. For the other compilers, you would have to explicitly define a macro to identify such a debug configuration.

See also

- *Using the indirection pattern for preprocessor stringification and concatenation*

Using the indirection pattern for preprocessor stringification and concatenation

The C++ preprocessor provides two operators for transforming identifiers into strings and concatenating identifiers together. The first one, `operator #`, is called the *stringizing* operator, and the second one, `operator ##`, is called the *token-pasting*, *merging*, or *concatenating* operator. Although their use is limited to some particular cases, it is important to understand how they work.

Getting ready

For this recipe, you need to know how to define macros using the preprocessor directive `#define`.

How to do it…

To create a string from an identifier using the preprocessor's `operator#`, use the following pattern:

1. Define a helper macro taking one argument that expands to `#` followed by the argument:

```
#define MAKE_STR2(x) #x
```

2. Define the macro you want to use, taking one argument that expands to the helper macro:

```
#define MAKE_STR(x) MAKE_STR2(x)
```

To concatenate identifiers together using the preprocessor's `operator##`, use the following pattern:

1. Define a helper macro with one or more arguments that use the token-pasting operator `##` to concatenate arguments:

```
#define MERGE2(x, y)    x##y
```

2. Define the macro you want to use, by using the helper macro:

```
#define MERGE(x, y)     MERGE2(x, y)
```

How it works…

To understand how these work, let's consider the `MAKE_STR` and `MAKE_STR2` macros defined earlier. When used with any text, they will produce a string containing that text. The following example shows how both these macros can be used to define strings containing the text *"sample"*.

```
std::string s1 { MAKE_STR(sample) };  // s1 = "sample"
std::string s2 { MAKE_STR2(sample) }; // s2 = "sample"
```

On the other hand, when a macro is passed as an argument, the results are different. In the following example, `NUMBER` is a macro that expands to an integer `42`. When used as an argument to `MAKE_STR`, it indeed produces the string *"42"*; however, when used as an argument to `MAKE_STR2`, it produces string *"NUMBER"*:

```
#define NUMBER 42

std::string s3 { MAKE_STR(NUMBER) };    // s3 = "42"
std::string s4 { MAKE_STR2(NUMBER) };   // s4 = "NUMBER"
```

The C++ standard defines the following rules for argument substitution in function-like macros (paragraph 16.3.1):

After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. A parameter in the replacement list, unless preceded by a # or ## preprocessing token or followed by a ## preprocessing token (see below), is replaced by the corresponding argument after all the macros contained therein have been expanded. Before being substituted, each argument's preprocessing tokens are completely macro replaced as if they formed the rest of the preprocessing file; no other preprocessing tokens is available.

What this says is that macro arguments are expanded before they are substituted into the macro body, except for the case when the operator `#` or `##` is preceding or following a parameter in the macro body. As a result, the following happens:

- For `MAKE_STR2(NUMBER)`, the `NUMBER` parameter in the replacement list is preceded by `#` and, therefore, it is not expanded before substituting the argument in the macro body; therefore, after the substitution, we have `#NUMBER` that becomes *"NUMBER"*.
- For `MAKE_STR(NUMBER)`, the replacement list is `MAKE_STR2(NUMBER)` that has no `#` or `##`; therefore, the `NUMBER` parameter is replaced with its corresponding argument, `42`, before being substituted. The result is `MAKE_STR2(42)`, which is then scanned again and, after expansion, it becomes *"42"*.

The same processing rules apply to macros using the token-pasting operator. Therefore, in order to make sure that your stringification and concatenation macros work for all cases, always apply the indirection pattern described in this recipe.

The token-pasting operator is typically used in macros that factor repetitive code to avoid

writing the same thing explicitly over and over again. The following simple example shows a practical use of the token-pasting operator; given a set of classes, we want to provide factory methods that create an instance of each class:

```
#define DECL_MAKE(x)    DECL_MAKE2(x)
#define DECL_MAKE2(x)   x* make##_##x() { return new x(); }

struct bar {};
struct foo {};

DECL_MAKE(foo)
DECL_MAKE(bar)

auto f = make_foo(); // f is a foo*
auto b = make_bar(); // b is a bar*
```

Those familiar with the Windows platform have probably used the `_T` (or `_TEXT`) macro for declaring string literals that are either translated to Unicode or ANSI strings (both single- and multi-type character strings):

```
auto text{ _T("sample") }; // text is either "sample" or L"sample"
```

The Windows SDK defines the `_T` macro as follows. Note that when `_UNICODE` is defined, the token-pasting operator is defined to concatenate together the `L` prefix and the actual string being passed to the macro:

```
#ifdef _UNICODE
#define __T(x)   L ## x
#else
#define __T(x)   x
#endif

#define _T(x)    __T(x)
#define _TEXT(x) __T(x)
```

At a first look, it seems unnecessary to have one macro calling another macro, but this level of indirection is key for making the `#` and `##` operators work with other macros, as we have seen in this recipe.

See also

- *Conditionally compiling your source code*

Performing compile-time assertion checks with static_assert

In C++, it is possible to perform both runtime and compile-time assertion checks to ensure that specific conditions in your code are true. Runtime assertions have the disadvantage that they are verified late when the program is running, and only if the control flow reaches them. There is no alternative when the condition depends on runtime data; however, when that is not the case, compile-time assertion checks are to be preferred. With compile-time assertions, the compiler is able to notify you early in the development stage with an error that a particular condition is not met. These, however, can only be used when the condition can be evaluated at compile time. In C++11, compile-time assertions are performed with `static_assert`.

Getting ready

The most common use of static assertion checks is with template metaprogramming, where they can be used for validating that preconditions on template types are met (examples can include whether a type is a POD type, copy-constructible, or a reference type, and so on). Another typical example is to ensure that types (or objects) have an expected size.

How to do it…

Use `static_assert` declarations to ensure that conditions in different scopes are met:

- *namespace*: In this example, we validate that the size of the class `item` is always 16:

```
struct alignas(8) item
{
  int      id;
  bool     active;
  double   value;
};

static_assert(sizeof(item) == 16,
              "size of item must be 16 bytes");
```

- *class*: In this example, we validate that `pod_wrapper` can only be used with POD types:

```
template <typename T>
class pod_wrapper
{
  static_assert(std::is_pod<T>::value,
                "POD type expected!");
  T value;
};

struct point
{
  int x;
  int y;
};
pod_wrapper<int>         w1; // OK
pod_wrapper<point>       w2; // OK
pod_wrapper<std::string> w3; // error: POD type expected
```

- *block (function)*: In this example, we validate that a function template has only arguments of an integral type:

```
template<typename T> auto
mul(T const a, T const b)
{
  static_assert(std::is_integral<T>::value,
                "Integral type expected");
  return a * b;
}

auto v1 = mul(1, 2);       // OK
auto v2 = mul(12.0, 42.5); // error: Integral type expected
```

How it works…

`static_assert` is basically a declaration, but it does not introduce a new name. These declarations have the following form:

```
    static_assert(condition, message);
```

The condition must be convertible to a Boolean value at compile time, and the message must be a string literal. As of C++17, the message is optional.

When the condition in a `static_assert` declaration evaluates to `true`, nothing happens. When the condition evaluates to `false`, the compiler generates an error that contains the specified message, if any.

See also

- *Conditionally compiling classes and functions with enable_if*
- *Selecting branches at compile time with constexpr if*

Conditionally compiling classes and functions with enable_if

Template metaprogramming is a powerful feature of C++ that enables us to write generic classes and functions that work with any type. That is actually a problem sometimes because the language does not define any mechanism for specifying constraints on the types that can be substituted for the template parameters. However, we can still achieve that using metaprogramming tricks and leveraging a rule called *substitution failure is not an error*, known shortly as *SFINAE*. This recipe will focus on implementing type constraints for templates.

Getting ready

Developers have used a class template usually called `enable_if` for many years in conjunction with SFINAE to implement constraints on template types. The `enable_if` family of templates has become part of the C++11 standard and is implemented as follows:

```
template<bool Test, class T = void>
struct enable_if
{};

template<class T>
struct enable_if<true, T>
{
  typedef T type;
};
```

To be able to use `std::enable_if`, you must include the header `<type_traits>`.

How to do it…

`std::enable_if` can be used in multiple scopes to achieve different purposes; consider the following examples:

- On a class template parameter to enable a class template only for types that meet a specified condition:

```
template <typename T,
          typename = typename
             std::enable_if<std::is_pod<T>::value, T>::type>
class pod_wrapper
{
  T value;
};

struct point
{
  int x;
  int y;
};

pod_wrapper<int>         w1; // OK
pod_wrapper<point>       w2; // OK
pod_wrapper<std::string> w3; // error: too few template arguments
```

- On a function template parameter, function parameter, or function return type to enable a function template only for types that meet a specified condition:

```
template<typename T,
         typename = typename std::enable_if<
            std::is_integral<T>::value, T>::type>
auto mul(T const a, T const b)
{
  return a * b;
}

auto v1 = mul(1, 2);     // OK
auto v2 = mul(1.0, 2.0);
// error: no matching overloaded function found
```

To simplify the cluttered code that we end up writing when we use `std::enable_if`, we can leverage alias templates and define two aliases, called `EnableIf` and `DisabledIf`:

```
template <typename Test, typename T = void>
using EnableIf = typename std::enable_if<Test::value, T>::type;

template <typename Test, typename T = void>
using DisableIf = typename std::enable_if<!Test::value, T>::type;
```

Based on these alias templates, the following definitions are equivalent to the ones shown above:

```
template <typename T, typename = EnableIf<std::is_pod<T>>>
class pod_wrapper
{
  T value;
};

template<typename T, typename = EnableIf<std::is_integral<T>>>
auto mul(T const a, T const b)
{
  return a * b;
}
```

How it works…

`std::enable_if` works because the compiler applies the SFINAE rule when performing overload resolution. Before we can explain how `std::enable_if` works, we should have a quick look at what SFINAE is.

When the compiler encounters a function call, it needs to build a set of possible overloads and select the best match for the call based on the arguments for the function call. When building this overload set, the compiler evaluates function templates too and has to perform a substitution for the specified or deduced types into the template arguments. According to SFINAE, when the substitution fails, instead of yielding an error, the compiler should just remove the function template from the overload set and continue.

The standard specifies a list of type and expression errors that are also SFINAE errors. These include an attempt to create an array of `void` or an array of size zero, an attempt to create a reference to `void`, an attempt to create a function type with a parameter of type `void`, or an attempt to perform an invalid conversion in a template argument expression or in an expression used in a function declaration. For the complete list of exceptions, consult the C++ standard or other resources.

Let's consider the following two overloads of a function called `func()`. The first overload is a function template that has a single argument of type `T::data_type`; that means it can only be instantiated with types that have an inner type called `data_type`. The second overload is a function that has a single argument of type `int`:

```
template <typename T>
void func(typename T::data_type const a)
{ std::cout << "func<>" << std::endl; }

void func(int const a)
{ std::cout << "func" << std::endl; }

template <typename T>
struct some_type
{
  using data_type = T;
};
```

If the compiler encounters a call such as `func(42)`, then it must find an overload that can take an `int` argument. When it builds the overload set and substitutes the template parameter with the provided template argument, the result `void func(int::data_type const)` is invalid, because `int` does not have a `data_type` member. Due to SFINAE, the compiler will not emit an error and stop, but will simply ignore the overload and continue. It then finds `void func(int const)`, and that will be the best (and only) match that it will call.

If the compiler encounters a call such as, `func<some_type<int>>(42)`, then it builds an overload set containing `void func(some_type<int>::data_type const>` and `void func(int const)`, and the best match in this case is the first overload; no SFINAE is involved this time.

On the other hand, if the compiler encounters a call such as, `func("string"s)`, then it again relies on SFINAE to ignore the function template, because `std::basic_string` does not have a

`value_type` member either. This time, however, the overload set does not contain any match for the string argument; therefore, the program is ill-formed and the compiler emits an error and stops.

The class template `enable_if<bool, T>` does not have any members, but its partial specialization `enable_if<true, T>` does have an inner type called `type`, that is a synonym for `T`. When the compile-time expression supplied as the first argument to `enable_if` evaluates to `true`, the inner member `type` is available, otherwise it is not.

Considering the last definition of function `mul()` from the *How to do it…* section, when the compiler encounters a call such as, `mul(1, 2)`, it tries to substitute `int` for the template parameter `T`; since `int` is an integral type, `std::is_integral<T>` evaluates to `true` and, therefore, a specialization of `enable_if` that defines an inner type called `type` is instantiated. As a result, the alias template `EnableIf` becomes a synonym for this type, which is `void` (from the expression `typename T = void`). The result is a function template `int mul<int, void>(int a, int b)` that can be called with the supplied arguments.

On the other hand, when the compiler encounters a call such as `mul(1.0, 2.0)`, it tries to substitute `double` for the template parameter `T`. However, this is not an integral type; as a result, the condition in `std::enable_if` evaluates to `false` and the class template does not define an inner member `type`. This results in a substitution error, but according to SFINAE, the compiler will not emit an error but move on. However, since no other overload is found, there will be no `mul()` function that can be called. Therefore, the program is considered ill-formed and the compiler stops with an error.

A similar situation is encountered with the class template `pod_wrapper`. It has two template type parameters: the first is the actual POD type that is being wrapped, and the second is the result of the substitution of `enable_if` and `is_pod`. If the type is a POD type (as in `pod_wrapper<int>`), then the inner member `type` from `enable_if` exists and it substitutes the second template type parameter. However, if the inner member `type` is not a POD type (as in `pod_wrapper<std::string>`), then the inner member `type` is not defined, and the substitution fails, producing an error such as *"too few template arguments"*.

There’s more…

`static_assert` and `std::enable_if` can be used to achieve the same goals. In fact, in the previous recipe, *Performing compile-time assertion checks with static_assert*, we have defined the same class template `pod_wrapper` and function template `mul()`. For these examples, `static_assert` seems like a better solution because the compiler emits better error messages (provided that you specify relevant messages in the `static_assert` declaration). These two, however, work quite differently and are not intended as alternatives.

`static_assert` does not rely on SFINAE and is applied after overload resolution is performed. The result of a failed assert is a compiler error. On the other hand, `std::enable_if` is used to remove candidates from the overload set and does not trigger compiler errors (given that the exceptions the standard specifies for SFINAE do not occur). The actual error that can occur after SFINAE is an empty overload set that makes a program ill-formed since a particular function call cannot be performed.

To understand the difference between `static_assert` and `std::enable_if` with SFINAE, let’s consider the case when we want to have two function overloads: one that should be called for arguments of integral types and one for arguments of any other type than integral types. With `static_assert`, we can write the following (note that the dummy second type parameter on the second overload is necessary to define two different overloads, otherwise we would just have two definitions of the same function):

```
template <typename T>
auto compute(T const a, T const b)
{
  static_assert(std::is_integral<T>,
                "An integral type expected");
  return a + b;
}

template <typename T, typename = void>
auto compute(T const a, T const b)
{
  static_assert(!std::is_integral<T>,
                "A non-integral type expected");
  return a * b;
}

auto v1 = compute(1, 2);
// error: ambiguous call to overloaded function

auto v2 = compute(1.0, 2.0);
// error: ambiguous call to overloaded function
```

Regardless of how we try to call this function, we end up with an error, because the compiler finds two overloads that it could potentially call. This is because `static_assert` is only considered after the overload resolution has been resolved, which, in this case, builds a set of two possible candidates.

The solution to this problem is `std::enable_if` and SFINAE. We use `std::enable_if` via the alias templates `EnableIf` and `DisableIf` defined above on a template parameter (although we still use the dummy template parameter on the second overload to introduce two different definitions). The following example shows the two overloads rewritten. The first overload is enabled only for integral types, while the second is disabled for integral types.

```
template <typename T, typename = EnableIf<std::is_integral<T>>>
auto compute(T const a, T const b)
{
  return a * b;
}

template <typename T, typename = DisableIf<std::is_integral<T>>,
          typename = void>
auto compute(T const a, T const b)
{
  return a + b;
}
auto v1 = compute(1, 2);     // OK; v1 = 2
auto v2 = compute(1.0, 2.0); // OK; v2 = 3.0
```

With SFINAE at work, when the compiler builds the overload set for either `compute(1, 2)` or `compute(1.0, 2.0)`; it will simply discard the overload that produces a substitution failure and move on, ending up in each case with an overload set containing a single candidate.

See also

- *Performing compile-time assertion checks with static_assert*
- *Creating type aliases and alias templates* recipe of Chapter 1, *Learning Modern Core Language Features*

Selecting branches at compile time with constexpr if

In the previous recipes, we saw how we can impose restrictions on types and functions using `static_assert` and `std::enable_if` and how these two are different. Template metaprogramming can become complicated and cluttered when we use SFINAE and `std::enable_if` to define function overloads or when we write variadic function templates. A new feature of C++17 is intended to simplify such code; it is called *constexpr if*, and it defines an `if` statement with a condition that is evaluated at compile time, resulting in the compiler selecting the body of a branch or another into the translation unit. Typical usage of constexpr if is for simplification of variadic templates and `std::enable_if`-based code.

Getting ready

In this recipe, we will refer to and simplify the code written in previous recipes. Before continuing with the recipe, you should take a moment to go back and review the code we have written in those recipes, which is as follows:

- The `compute()` overloads for integral and non-integral types from the *Conditionally compiling classes and functions with enable_if* recipe.
- User-defined 8-, 16-, and 32-bit binary literals from the *Creating raw user-defined literals* recipe of Chapter 2, *Working with Numbers and Strings*.

These implementations have several issues:

- They are hard to read. There is a lot of focus on the template declaration, yet the body of the functions are very simple, for instance. The biggest problem, though, is that it requires a greater attention from developers because it is cluttered with complicated declarations, such as `typename = std::enable_if<std::is_integral<T>::value, T>::type`.
- There is too much code. The end purpose in the first example is to have a generic function that behaves differently for different types, yet we had to write two overloads for the function; moreover, to differentiate the two, we had to use an extra, unused, template parameter. In the second example, the purpose was to build an integer value out of characters `'0'` and `'1'`, yet we had to write one class template and three specializations to make it happen.
- It requires advanced template metaprogramming skills, which shouldn't be necessary for doing something this simple.

The syntax for constexpr if is very similar to regular `if` statements and requires the `constexpr` keyword before the condition. The general form is as follows:

```
if constexpr (init-statement condition) statement-true
else statement-false
```

How to do it…

Use constexpr if statements to do the following:

- To avoid using `std::enable_if` and relying on SFINAE to impose restrictions on function template types and conditionally compile code:

```
template <typename T>
auto compute(T const a, T const b)
{
  if constexpr (std::is_integral<T>::value)
    return a * b;
  else
    return a + b;
}
```

- To simplify writing variadic templates and implement metaprogramming compile-time recursion:

```
namespace binary
{
  using byte8 = unsigned char;

  namespace binary_literals
  {
    namespace binary_literals_internals
    {
      template <typename CharT, char d, char... bits>
      constexpr CharT binary_eval()
      {
        if constexpr(sizeof...(bits) == 0)
          return static_cast<CharT>(d-'0');
        else if constexpr(d == '0')
          return binary_eval<CharT, bits...>();
        else if constexpr(d == '1')
          return static_cast<CharT>(
            (1 << sizeof...(bits)) |
            binary_eval<CharT, bits...>());
      }
    }

    template<char... bits>
    constexpr byte8 operator""_b8()
    {
      static_assert(
        sizeof...(bits) <= 8,
        "binary literal b8 must be up to 8 digits long");

      return binary_literals_internals::
                  binary_eval<byte8, bits...>();
    }
  }
}
```

How it works…

The way constexpr if works is relatively simple: the condition in the `if` statement must be a compile-time expression that evaluates or can be convertible to a Boolean. If the condition is `true`, the body of the `if` statement is selected, which means it ends up into the translation unit for compilation. If the condition is `false`, the `else` branch, if any is defined, is evaluated. Return statements in discarded constexpr if branches to not contribute to the function return type deduction.

In the first example from the *How to do it…* section, the `compute()` function template has a clean signature. The body is also very simple; if the type that is substituted for the template parameter is an integral type, the compiler will select the first branch (that is, `return a * b;`) for code generation and discard the `else` branch. For non-integral types, because the condition evaluates to `false`, the compiler will select the `else` branch (that is, `return a + b;`) for code generation and discard the rest.

In the second example from the *How to do it…* section, the internal helper function `binary_eval()` is a variadic template function without any parameters; it only has template parameters. The function evaluates the first argument and then does something with the rest of the arguments in a recursive manner (but remember this is not a runtime recursion). When there is a single character left and the size of the remaining pack is 0, we return the decimal value represented by the character (0 for `'0'` and 1 for `'1'`). If the current first element is a `'0'`, we return the value determined by evaluating the rest of the arguments pack, which involves a recursive call. If the current first element is a `'1'`, we return the value by shifting a 1 to the left on a number of positions given by the size of the remaining pack bit or the value determined by evaluating the rest of the arguments pack, which again involves a recursive call.

Providing metadata to the compiler with attributes

C++ has been very deficient when it comes to features that enable reflection or introspection on types or data or standard mechanisms to define language extensions. Because of that, compilers have defined their own specific extensions for this purpose. Examples include the VC++ `__declspec()` specifier or the GCC `__attribute__((...))`. C++11, however, introduces the concept of attributes that enable compilers to implement extensions in a standard way or even embedded domain-specific languages. The new C++ standards define several attributes all compilers should implement, and that will be the topic of this recipe.

How to do it…

Use standard attributes to provide hints for the compiler about various design goals:

- To ensure that the return value from a function cannot be ignored, declare the function with the `[[nodiscard]]` attribute:

```
[[nodiscard]] int get_value1()
{
  return 42;
}

get_value1();
// warning: ignoring return value of function
//          declared with 'nodiscard' attribute get_value1();
```

- Alternatively, you can declare enumerations and classes used as the return type of a function with the `[[nodiscard]]` attribute; in this case, the return value of any function returning such a type cannot be ignored:

```
enum class[[nodiscard]] ReturnCodes{ OK, NoData, Error };

ReturnCodes get_value2()
{
  return ReturnCodes::OK;
}

struct[[nodiscard]] Item{};

Item get_value3()
{
  return Item{};
}

// warning: ignoring return value of function
//          declared with 'nodiscard' attribute
get_value2();
get_value3();
```

- To ensure that usage of functions or types that are considered deprecated is flagged by the compiler with a warning, declare them with the `[[deprecated]]` attribute:

```
[[deprecated("Use func2()")]] void func()
{
}

// warning: 'func' is deprecated : Use func2()
func();

class [[deprecated]] foo
{
};

// warning: 'foo' is deprecated
foo f;
```

- To ensure that the compiler does not emit a warning for unused variables, use the `[[maybe_unused]]` attribute:

```
double run([[maybe_unused]] int a, double b)
{
  return 2 * b;
```

```
}

[[maybe_unused]] auto i = get_value1();
```

- To ensure that intentional fall-through case labels in a `switch` statement are not flagged by the compiler with a warning, use the `[[fallthrough]]` attribute:

```
void option1() {}
void option2() {}

int alternative = get_value1();
switch (alternative)
{
  case 1:
    option1();
    [[fallthrough]]; // this is intentional
  case 2:
   option2();
}
```

How it works…

The attributes are a very flexible feature of C++; they can be used almost everywhere, but the actual usage is specifically defined for each particular attribute. They can be used on types, functions, variables, names, code blocks, or entire translation units.

Attributes are specified between double square brackets (for example `[[attr1]]`) and more than one attribute can be specified in a declaration (for example, `[[attr1, attr2, attr3]]`).

Attributes can have arguments, for example, `[[mode(greedy)]]`, and can be fully qualified, for example, `[[sys::hidden]]` or `[[using sys: visibility(hidden), debug]]`.

Attributes can appear either before or after the name of the entity on which they are applied, or both, in which case they are combined. The following are several examples that exemplify this:

```
// attr1 applies to a, attr2 applies to b
int a [[attr1]], b [[attr2]];
// attr1 applies to a and b
int [[attr1]] a, b;
// attr1 applies to a and b, attr2 applies to a
int [[attr1]] a [[attr2]], b;
```

Attributes cannot appear in a namespace declaration, but they can appear as a single line declaration anywhere in a namespace. In this case, it is specific to each attribute whether it applies to the following declaration, to the namespace, or to the translation unit:

```
namespace test
{
  [[debug]];
}
```

Attributes are often ignored or briefly mentioned in books and tutorials on modern C++ programming, and the reason for that is probably the fact that developers cannot actually write attributes, as this language feature is intended for compiler implementations. For some compilers, it may be possible, though, to define user-provided attributes; such a compiler is GCC, which supports plugins that add extra features to the compiler, and they can be used for defining new attributes too.

However, the standard does define several attributes all compilers must implement and using them can help you write better code. We have seen some of them in the examples from the previous section. These attributes have been defined in different versions of the standard:

- In C++11:
  - The `[[noreturn]]` attribute indicates that a function does not return.
  - The `[[carries_dependency]]` attribute indicates that the dependency chain in release-consume `std::memory_order` propagates in and out of the function, which allows the compiler to skip unnecessary memory fence instructions.
- In C++14:
  - The `[[deprecated]]` and `[[deprecated("reason")]]` attributes indicate that the entity declared with these attributes is considered deprecated and should not be used.

These attributes can be used with classes, non-static data members, typedefs, functions, enumerations, and template specializations. The reason string is an optional parameter.

- In C++17:
  - The `[[fallthrough]]` attribute indicates that fall-through between labels in a switch statement is intentional. The attribute must appear on a line of its own immediately before a case label.
  - The `[[nodiscard]]` attribute indicates that a return value from a function cannot be ignored.
  - The `[[maybe_unused]]` attribute indicates that an entity may be unused, but the compiler should not emit a warning about that. This attribute can be applied to variables, classes, non-static data members, enumerations, enumerators, and typedefs.

Standard Library Containers, Algorithms, and Iterators

We will cover the following recipes in this chapter:

- Using vector as a default container
- Using bitset for fixed-size sequences of bits
- Using vector<bool> for variable-size sequences of bits
- Finding elements in a range
- Sorting a range
- Initializing a range
- Using set operations on a range
- Using iterators to insert new elements in a container
- Writing your own random access iterator
- Container access with non-member functions

Introduction

The C++ standard library has evolved a lot with C++11, C++14 and C++17. However, at its core, initially sat three main pillars: containers, algorithms, and iterators. They are all implemented as general purpose classes or function templates. In this chapter, we'll look at how these could be employed together for achieving various goals.

Using vector as a default container

The standard library provides various types of containers that store collections of objects; the library includes sequence containers (such as `vector`, `array`, or `list`), ordered and unordered associative containers (such as `set` and `map`), and container adapters that do not store data but provide an adapted interface towards a sequence container (such as `stack` and `queue`). All of them are implemented as class templates, which means they can be used with any type (providing it meets the container requirements). Though you should always use the container that is the most appropriate for a particular problem (which not only provides good performance in terms of speed of inserts, deletes, access to elements, and memory usage but also makes the code easy to read and maintain), the default choice should be `vector`. In this recipe, we will see why `vector` should be the preferred choice for a container and what are the most common operations with `vector`.

Getting ready

The reader is expected to be familiar with C-like arrays, both statically and dynamically allocated.

The class template `vector` is available in the `std` namespace in the `<vector>` header.

How to do it…

To initialize a `std::vector` class template, you can use any of the following methods, but you are not restricted to only these:

- Initialize from an initialization list:

```
std::vector<int> v1 { 1, 2, 3, 4, 5 };
```

- Initialize from a C-like array:

```
int arr[] = { 1, 2, 3, 4, 5 };
std::vector<int> v2(arr, arr + 5); // { 1, 2, 3, 4, 5 }
```

- Initialize from another container:

```
std::list<int> l{ 1, 2, 3, 4, 5 };
std::vector<int> v3(l.begin(), l.end()); //{ 1, 2, 3, 4, 5 }
```

- Initialize from a count and a value:

```
std::vector<int> v4(5, 1); // {1, 1, 1, 1, 1}
```

To modify the content of `std::vector`, use any of the following methods, but you are not restricted to only these:

- Add an element at the end of the vector with `push_back()`:

```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
v1.push_back(6); // v1 = { 1, 2, 3, 4, 5, 6 }
```

- Remove an element from the end of the vector with `pop_back()`:

```
v1.pop_back();
```

- Insert anywhere in the vector with `insert()`:

```
int arr[] = { 1, 2, 3, 4, 5 };
std::vector<int> v2;
v2.insert(v2.begin(), arr, arr + 5); // v2 = { 1, 2, 3, 4, 5 }
```

- Add an element by creating it at the end of the vector with `emplace_back()`:

```
struct foo
{
  int a;
  double b;
  std::string c;

  foo(int a, double b, std::string const & c) :
    a(a), b(b), c(c) {}
};

std::vector<foo> v3;
v3.emplace_back(1, 1.0, "one"s);
// v3 = { foo{1, 1.0, "one"} }
```

- Insert an element by creating it anywhere in the vector with `emplace()`:

```
v3.emplace(v3.begin(), 2, 2.0, "two"s);
// v3 = { foo{2, 2.0, "two"}, foo{1, 1.0, "one"} }
```

To modify the whole content of the vector, use any of the following methods, but you are not restricted to only these:

- Assign from another vector with `operator=`; this replaces the content of the container:

```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
std::vector<int> v2{ 10, 20, 30 };
v2 = v1; // v1 = { 1, 2, 3, 4, 5 }
```

- Assign from another sequence defined by a begin and end iterator with the `assign()` method; this replaces the content of the container:

```
int arr[] = { 1, 2, 3, 4, 5 };
std::vector<int> v3;
v3.assign(arr, arr + 5); // v3 = { 1, 2, 3, 4, 5 }
```

- Swap the content of two vectors with the `swap()` method:

```
std::vector<int> v4{ 1, 2, 3, 4, 5 };
std::vector<int> v5{ 10, 20, 30 };
v4.swap(v5); // v4 = { 10, 20, 30 }, v5 = { 1, 2, 3, 4, 5 }
```

- Remove all the elements with the `clear()` method:

```
std::vector<int> v6{ 1, 2, 3, 4, 5 };
v6.clear(); // v6 = { }
```

- Remove one or more elements with the `erase()` method (which requires either an iterator or a pair of iterators that define the range of elements from the vector to be removed):

```
std::vector<int> v7{ 1, 2, 3, 4, 5 };
v7.erase(v7.begin() + 2, v7.begin() + 4); // v7 = { 1, 2, 5 }
```

To get the address of the first element in a vector, usually to pass the content of a vector to a C-like API, use any of the following methods:

- Use the `data()` method, which returns a pointer to the first element, providing direct access to the underlying contiguous sequence of memory where the vector elements are stored; this is only available since C++11:

```
void process(int const * const arr, int const size)
{ /* do something */ }

std::vector<int> v{ 1, 2, 3, 4, 5 };
process(v.data(), static_cast<int>(v.size()));
```

- Get the address of the first element:

```
process(&v[0], static_cast<int>(v.size()));
```

- Get the address of the element referred by the `front()` method:

```
        process(&v.front(), static_cast<int>(v.size()));
```

- Get the address of the element pointed by the iterator returned from `begin()`:

```
        process(&*v.begin(), static_cast<int>(v.size()));
```

How it works…

The `std::vector` class is designed to be the C++ container most similar to and inter-operable with C-like arrays. A vector is a variable-sized sequence of elements, guaranteed to be stored contiguously in memory, which makes the content of a vector easily passable to a C-like function that takes a pointer to an element of an array and, usually, a size. There are many benefits of using a vector instead of C-like arrays and these benefits include:

- No direct memory management is required from the developer, as the container does this internally, allocating memory, reallocating, and releasing.

Note that a vector is intended for storing object instances. If you need to store pointers, do not store raw pointers but smart pointers. Otherwise, you need to handle the lifetime management of the pointed objects.

- The possibility of modifying the size of the vector.
- Simple assignment or concatenation of two vectors.
- Direct comparison of two vectors.

The `vector` class is a very efficient container, with all implementations providing a lot of optimizations that most developers are not capable of doing with C-like arrays. Random access to its elements and insertion and removal at the end of a vector is a constant *O(1)* operation (provided that reallocation is not necessary), while insertion and removal anywhere else is a linear *O(n)* operation.

Compared to other standard containers, the vector has various benefits:

- It is compatible with C-like arrays and C-like APIs; the content of other containers (except for `std::array`) needs to be copied to a vector before being passed to a C-like API expecting an array.
- It has the fastest access to elements of all containers.
- It has no per-element memory overhead for storing elements, as elements are stored in a contiguous space, like a C array (and unlike other containers such as `list` that requires additional pointers to other elements, or associative containers that require hash values).

`std::vector` is very similar in semantics to C-like arrays but has a variable size. The size of a vector can increase and decrease. There are two properties that define the size of a vector:

- *Capacity* is the number of elements the vector can accommodate without performing additional memory allocations; this is indicated by the `capacity()` method.
- *Size* is the actual number of elements in the vector; this is indicated by the `size()` method.

Size is always smaller or equal to capacity. When size is equal to capacity and a new

element needs to be added, the capacity needs to be modified so that the vector has space for more elements. In this case, the vector allocates a new chunk of memory and moves the previous content to the new location and then frees the previously allocated memory. Though this sounds time-consuming (and it is), implementations increase the capacity exponentially, by doubling it each time it needs to be changed. As a result, on average, each element of the vector only needs to be moved once (that is because all the elements of the vector are moved during an increase of capacity, but then an equal number of elements can be added without incurring more moves, given that insertions are performed at the end of the vector).

If you know beforehand how many elements will be inserted in the vector, you can first call the `reserve()` method to increase the capacity to at least the specified amount (this method does nothing if the specified size is smaller than the current capacity) and only then insert the elements.

On the other hand, if you need to free additional reserved memory, you can use the `shrink_to_fit()` method to request this, but it is an implementation decision whether to free any memory or not. An alternative to this non-binding method, available since C++11, is to do a swap with a temporary, empty vector:

```
std::vector<int> v{ 1, 2, 3, 4, 5 };
std::vector<int>().swap(v); // v.size = 0, v.capacity = 0
```

Calling the `clear()` method only removes all the elements from the vector but does not free any memory.

It should be noted that the vector implements operations specific to other types of containers:

- `stack`: With `push_back()` and `emplace_back()` to add at the end and `pop_back()` to remove from the end. Keep in mind that `pop_back()` does not return the last element that has been removed. You need to access that explicitly, if that is necessary, for instance, using the `back()` method before removing the element.
- `list`: With `insert()` and `emplace()` to add elements in the middle of the sequence and `erase()` to remove elements from anywhere in the sequence.

There’s more…

The rule of thumb for C++ containers is: use `std::vector` *as the default container unless you have good reasons to use another one.*

See also

- *Using bitset for fixed-size sequences of bits*
- *Using vector<bool> for variable-size sequences of bits*

Using bitset for fixed-size sequences of bits

It is not uncommon for developers to operate with bit flags; this can be either because they work with operating system APIs, usually written in C, that take various types of arguments (such as options or styles) in the form of bit flags, or because they work with libraries that do similar things, or simply because some types of problems are naturally solved with bit flags. One can think of alternatives to working with bits and bit operations, such as defining arrays having one element for every option/flag, or defining a structure with members and functions to model the bit flags, but these are often more complicated, and in case you need to pass a numerical value representing bit flags to a function you still need to convert the array or the structure to a sequence of bits. For this reason, the C++ standard provides a container called `std::bitset` for fixed-size sequences of bits.

Getting ready

For this recipe, you must be familiar with bitwise operations (and, or, xor, not, and shifting).

The `bitset` class is available in the `std` namespace in the `<bitset>` header. A bitset represents a fixed-size sequence of bits, with the size defined at compile time. For convenience, in this recipe, all examples will be with bitsets of 8 bits.

How to do it…

To construct an `std::bitset` object, use one of the available constructors:

- An empty bitset with all bits set to 0:

```
std::bitset<8> b1; // [0,0,0,0,0,0,0,0]
```

- A bitset from a numerical value:

```
std::bitset<8> b2{ 10 }; // [0,0,0,0,1,0,1,0]
```

- A bitset from a string of `'0'` and `'1'`:

```
std::bitset<8> b3{ "1010"s }; // [0,0,0,0,1,0,1,0]
```

- A bitset from a string containing any two characters representing `'0'` and `'1'`; in this case, we must specify which character represents a 0 and which character represents a 1:

```
std::bitset<8> b4
  { "ooooxoxo"s, 0, std::string::npos, 'o', 'x' };
  // [0,0,0,0,1,0,1,0]
```

To test individual bits in the set or the entire set for specific values, use any of the available methods:

- `count()` to get the number of bits set to 1:

```
std::bitset<8> bs{ 10 };
std::cout << "has " << bs.count() << " 1s" << std::endl;
```

- `any()` to check whether there is at least one bit set to 1:

```
if (bs.any()) std::cout << "has some 1s" << std::endl;
```

- `all()` to check whether all the bits are set to 1:

```
if (bs.all()) std::cout << "has only 1s" << std::endl;
```

- `none()` to check whether all the bits are set to 0:

```
if (bs.none()) std::cout << "has no 1s" << std::endl;
```

- `test()` to check the value of an individual bit:

```
if (!bs.test(0)) std::cout << "even" << std::endl;
```

- `operator[]` to access and test individual bits:

```
if(!bs[0]) std::cout << "even" << std::endl;
```

To modify the content of a bitset, use any of the following methods:

- Member operators `|=`, `&=`, `^=` , and `~` to perform binary or, and, xor, and not operations, or non-member operators `|`, `&`, and `^`:

```
std::bitset<8> b1{ 42 }; // [0,0,1,0,1,0,1,0]
std::bitset<8> b2{ 11 }; // [0,0,0,0,1,0,1,1]
auto b3 = b1 | b2;       // [0,0,1,0,1,0,1,1]
auto b4 = b1 & b2;       // [0,0,0,0,1,0,1,0]
auto b5 = b1 ^ b2;       // [1,1,0,1,1,1,1,0]
auto b6 = ~b1;           // [1,1,0,1,0,1,0,1]
```

- Member operators `<<=`, `<<`, `>>=`, `>>` to perform shifting operations:

```
auto b7 = b1 << 2;       // [1,0,1,0,1,0,0,0]
auto b8 = b1 >> 2;       // [0,0,0,0,1,0,1,0]
```

- `flip()` to toggle the entire set or an individual bit from 0 to 1 or from 1 to 0:

```
b1.flip();               // [1,1,0,1,0,1,0,1]
b1.flip(0);              // [1,1,0,1,0,1,0,0]
```

- `set()` to change the entire set or an individual bit to `true` or the specified value:

```
b1.set(0, true);         // [1,1,0,1,0,1,0,1]
b1.set(0, false);        // [1,1,0,1,0,1,0,0]
```

- `reset()` to change the entire set or an individual bit to false:

```
b1.reset(2);             // [1,1,0,1,0,0,0,0]
```

To convert a bitset to a numerical or string value, use the following methods:

- `to_ulong()` and `to_ullong()` to convert to `unsigned long` or `unsigned long long`:

```
std::bitset<8> bs{ 42 };
auto n1 = bs.to_ulong();  // n1 = 42UL
auto n2 = bs.to_ullong(); // n2 = 42ULL
```

- `to_string()` to convert to `std::basic_string`; by default the result is a string containing `'0'` and `'1'`, but you can specify a different character for these two values:

```
auto s1 = bs.to_string();         // s1 = "00101010"
auto s2 = bs.to_string('o', 'x'); // s2 = "ooxoxoxo"
```

How it works…

If you've ever worked with C or C-like APIs, chances are you either wrote or at least have seen code that manipulates bits to define styles, options, or other kinds of values. This usually involves operations, such as:

- Defining the bit flags; these can be enumerations, static constants in a class, or macros introduced with `#define` in the C style. Usually, there is a flag representing no value (style, option, and so on). Since these are supposed to be bit flags, their values are powers of 2.
- Adding and removing flags from the set (that is, a numerical value). Adding a bit flag is done with the bit-or operator (`value |= FLAG`) and removing a bit flag is done with the bit-and operator, with the negated flag (`value &= ~FLAG`).
- Testing whether a flag is added to the set (`value & FLAG == FLAG`).
- Calling functions with the flags as an argument.

The following shows a simple example of flags defining the border style of a control that can have a border on the left, right, top, or bottom side, or any combination of these, including no border:

```
#define BORDER_NONE   0x00
#define BORDER_LEFT   0x01
#define BORDER_TOP    0x02
#define BORDER_RIGHT  0x04
#define BORDER_BOTTOM 0x08

void apply_style(unsigned int const style)
{
  if (style & BORDER_BOTTOM) { /* do something */ }
}

// initialize with no flags
unsigned int style = BORDER_NONE;
// set a flag
style = BORDER_BOTTOM;
// add more flags
style |= BORDER_LEFT | BORDER_RIGHT | BORDER_TOP;
// remove some flags
style &= ~BORDER_LEFT;
style &= ~BORDER_RIGHT;
// test if a flag is set
if ((style & BORDER_BOTTOM) == BORDER_BOTTOM) {}
// pass the flags as argument to a function
apply_style(style);
```

The standard `std::bitset` class is intended as a C++ alternative to this C-like working style with sets of bits. It enables us to write more robust and safer code because it abstracts the bit operations with member functions, though we still need to identify what each bit in the set is representing:

- Adding and removing flags is done with the `set()` and `reset()` methods, which set the value of a bit indicated by its position to 1 or 0 (or `true` and `false`); alternatively, we can use the index operator for the same purpose.
- Testing if a bit is set is done with the `test()` method.
- Conversion from an integer or a string is done through the constructor, and

conversion to an integer or string is done with member functions so that the values from bitsets can be used where integers are expected (such as arguments to functions).

In addition to these mentioned operations, the `bitset` class has additional methods for performing bitwise operations on bits, shifting, testing, and others that have been shown in the previous section.

Conceptually, `std::bitset` is a representation of a numerical value that enables you to access and modify individual bits. Internally, however, a bitset has an array of integer values on which it performs bit operations. The size of a bitset is not limited to the size of a numerical type; it can be anything, except that it is a compile-time constant.

The example with the control border styles from the previous section can be written using `std::bitset` in the following manner:

```
struct border_flags
{
  static const int left = 0;
  static const int top = 1;
  static const int right = 2;
  static const int bottom = 3;
};

// initialize with no flags
std::bitset<4> style;
// set a flag
style.set(border_flags::bottom);
// set more flags
style
  .set(border_flags::left)
  .set(border_flags::top)
  .set(border_flags::right);
// remove some flags
style[border_flags::left] = 0;
style.reset(border_flags::right);
// test if a flag is set
if (style.test(border_flags::bottom)) {}
// pass the flags as argument to a function
apply_style(style.to_ulong());
```

There’s more…

The bitset can be created from an integer and can convert its value to an integer using the `to_ulong()` or `to_ullong()` methods. However, if the size of the bitset is larger than the size of these numerical types and any of the bits beyond the size of the requested numerical type is set to `1`, then these methods throw an `std::overflow_error` exception because the value cannot be represented on `unsigned long` or `unsigned long long`. In order to extract all the bits, we need to do the following operations, as shown in the next code:

- Clear the bits beyond the size of `unsigned long` or `unsigned long long`.
- Convert the value to `unsigned long` or `unsigned long long`.
- Shift the bitset with the number of bits in `unsigned long` or `unsigned long long`.
- Do this until all the bits are retrieved.

```
template <size_t N>
std::vector<unsigned long> bitset_to_vectorulong(std::bitset<N> bs)
{
  auto result = std::vector<unsigned long> {};
  auto const size = 8 * sizeof(unsigned long);
  auto const mask = std::bitset<N>{ static_cast<unsigned long>(-1)};

  auto totalbits = 0;
  while (totalbits < N)
  {
    auto value = (bs & mask).to_ulong();
    result.push_back(value);
    bs >>= size;
    totalbits += size;
  }

  return result;
}

std::bitset<128> bs =
       (std::bitset<128>(0xFEDC) << 96) |
       (std::bitset<128>(0xBA98) << 64) |
       (std::bitset<128>(0x7654) << 32) |
       std::bitset<128>(0x3210);

std::cout << bs << std::endl;

auto result = bitset_to_vectorulong(bs);
for (auto const v : result)
  std::cout << std::hex << v << std::endl;
```

For cases where the size of the `bitset` cannot be known at compile time, the alternative is `std::vector<bool>`, which we will cover in the next recipe.

See also

- *Using vector<bool> for variable-size sequences of bits*

Using vector<bool> for variable-size sequences of bits

In the previous recipe, we looked at using `std::bitset` for fixed-size sequences of bits. Sometimes, however, an `std::bitset` is not a good choice because you do not know the number of bits at compile time, and just defining a set of a large enough number of bits is not a good idea because you can get into a situation when the number is not actually large enough. The standard alternative for this is to use the `std::vector<bool>` container that is a specialization of `std::vector` with space and speed optimizations, as implementations do not actually store Boolean values, but individual bits for each element.

For this reason, however, `std::vector<bool>` does not meet the requirements of a standard container or sequential container, nor does `std::vector<bool>::iterator` meet the requirements of a forward iterator. As a result, this specialization cannot be used in generic code where a vector is expected. On the other hand, being a vector, it has a different interface from that of `std::bitset` and cannot be viewed as a binary representation of a number. There are no direct ways to construct `std::vector<bool>` from a number or string nor to convert to a number or string.

Getting ready…

This recipe assumes you are familiar with both `std::vector` and `std::bitset`. If you didn’t read the previous recipes, *Using vector as a default container* and *Using bitset for fixed-size sequences of bits*, you should do that before continuing.

The `vector<bool>` class is available in the `std` namespace in the `<vector>` header.

How to do it…

To manipulate an `std::vector<bool>`, use the same methods you would use for an `std::vector<T>`, as shown in the following examples:

- Creating an empty vector:

```
std::vector<bool> bv; // []
```

- Adding bits to the vector:

```
bv.push_back(true);  // [1]
bv.push_back(true);  // [1, 1]
bv.push_back(false); // [1, 1, 0]
bv.push_back(false); // [1, 1, 0, 0]
bv.push_back(true);  // [1, 1, 0, 0, 1]
```

- Setting the values of individual bits:

```
bv[3] = true;        // [1, 1, 0, 1, 1]
```

- Using generic algorithms:

```
auto count_of_ones = std::count(bv.cbegin(), bv.cend(), true);
```

- Removing bits from the vector:

```
bv.erase(bv.begin() + 2); // [1, 1, 1, 1]
```

How it works…

`std::vector<bool>` is not a standard vector because it is designed to provide space optimization by storing a single bit for each element instead of a Boolean value. Therefore, its elements are not stored in a contiguous sequence and cannot be substituted for an array of Booleans. Due to this:

- The index operator cannot return a reference to a specific element because elements are not stored individually:

```
std::vector<bool> bv;
bv.resize(10);
auto& bit = bv[0];      // error
```

- Dereferencing an iterator cannot produce a reference to `bool` for the same reason as mentioned earlier:

```
auto& bit = *bv.begin(); // error
```

- There is no guarantee that individual bits can be manipulated independently at the same time from different threads.
- The vector cannot be used with algorithms that require forward iterators, such as `std::search()`.
- The vector cannot be used in some generic code where `std::vector<T>` is expected if such code requires any of the operations mentioned in this list.

An alternative to `std::vector<bool>` is `std::dequeu<bool>`, which is a standard container (a double-ended queue) that meets all container and iterator requirements and can be used with all standard algorithms. However, this will not have the space optimization that `std::vector<bool>` is providing.

There’s more…

The `std::vector<bool>` interface is very different from `std::bitset`. If you want to be able to write code in a similar manner, you can create a wrapper on `std::vector<bool>`, which looks like `std::bitset`, where possible. The following implementation provides members similar to what is available in `std::bitset`:

```
class bitvector
{
  std::vector<bool> bv;
public:
  bitvector(std::vector<bool> const & bv) : bv(bv) {}
  bool operator[](size_t const i) { return bv[i]; }

  inline bool any() const {
    for (auto b : bv) if (b) return true;
      return false;
  }

  inline bool all() const {
    for (auto b : bv) if (!b) return false;
      return true;
  }

  inline bool none() const { return !any(); }

  inline size_t count() const {
    return std::count(bv.cbegin(), bv.cend(), true);
  }

  inline size_t size() const { return bv.size(); }

  inline bitvector & add(bool const value) {
    bv.push_back(value);
    return *this;
  }

  inline bitvector & remove(size_t const index) {
    if (index >= bv.size())
      throw std::out_of_range("Index out of range");
    bv.erase(bv.begin() + index);
    return *this;
  }

  inline bitvector & set(bool const value = true) {
    for (size_t i = 0; i < bv.size(); ++i)
      bv[i] = value;
    return *this;
  }

  inline bitvector& set(size_t const index, bool const value = true) {
    if (index >= bv.size())
      throw std::out_of_range("Index out of range");
    bv[index] = value;
    return *this;
  }

  inline bitvector & reset() {
    for (size_t i = 0; i < bv.size(); ++i) bv[i] = false;
    return *this;
  }

  inline bitvector & reset(size_t const index) {
    if (index >= bv.size())
      throw std::out_of_range("Index out of range");
    bv[index] = false;
    return *this;
  }

  inline bitvector & flip() {
```

```
        bv.flip();
        return *this;
      }

      std::vector<bool>& data() { return bv; }
    };
```

This is only a basic implementation, and if you want to use such a wrapper, you should add additional methods, such as bit logic operations, shifting, maybe reading and writing from and to streams, and so on. However, with the preceding code, we can write the following examples:

```
    bitvector bv;
    bv.add(true).add(true).add(false); // [1, 1, 0]
    bv.add(false);                     // [1, 1, 0, 0]
    bv.add(true);                      // [1, 1, 0, 0, 1]

    if (bv.any()) std::cout << "has some 1s" << std::endl;
    if (bv.all()) std::cout << "has only 1s" << std::endl;
    if (bv.none()) std::cout << "has no 1s" << std::endl;
    std::cout << "has " << bv.count() << " 1s" << std::endl;

    bv.set(2, true);                   // [1, 1, 1, 0, 1]
    bv.set();                          // [1, 1, 1, 1, 1]

    bv.reset(0);                       // [0, 1, 1, 1, 1]
    bv.reset();                        // [0, 0, 0, 0, 0]

    bv.flip();                         // [1, 1, 1, 1, 1]
```

See also

- *Using vector as a default container*
- *Using bitset for fixed-size sequences of bits*

Finding elements in a range

One of the most common operations we do in any application is searching through data. Therefore, it is not surprising that the standard library provides many generic algorithms for searching through standard containers or anything that can represent a range and is defined by a start and a past-the-end iterator. In this recipe, we will see what these standard algorithms are and how they can be used.

Getting ready

For all the examples in this recipe, we will use `std::vector`, but all algorithms work with ranges defined by a begin and past-the-end, either input or forward iterators, depending on the algorithm (for more information about the various types of iterators, see the recipe, *Writing your own random access iterator*). All these algorithms are available in the `std` namespace in the `<algorithm>` header.

How to do it…

The following is a list of algorithms that can be used for finding elements in a range:

- Use `std::find()` to find a value in a range; this algorithm returns an iterator to the first element equal to the value:

```
std::vector<int> v{ 1, 1, 2, 3, 5, 8, 13 };

auto it = std::find(v.cbegin(), v.cend(), 3);
if (it != v.cend()) std::cout << *it << std::endl;
```

- Use `std::find_if()` to find a value in a range that meets a criterion from a unary predicate; this algorithm returns an iterator to the first element for which the predicate returns `true`:

```
std::vector<int> v{ 1, 1, 2, 3, 5, 8, 13 };

auto it = std::find_if(v.cbegin(), v.cend(),
                       [](int const n) {return n > 10; });
if (it != v.cend()) std::cout << *it << std::endl;
```

- Use `std::find_if_not()` to find a value in a range that does not meet a criterion from a unary predicate; this algorithm returns an iterator to the first element for which the predicate returns `false`:

```
std::vector<int> v{ 1, 1, 2, 3, 5, 8, 13 };

auto it = std::find_if_not(v.cbegin(), v.cend(),
                   [](int const n) {return n % 2 == 1; });
if (it != v.cend()) std::cout << *it << std::endl;
```

- Use `std::find_first_of()` to search for the occurrence of any value from a range in another range; this algorithm returns an iterator to the first element that is found:

```
std::vector<int> v{ 1, 1, 2, 3, 5, 8, 13 };
std::vector<int> p{ 5, 7, 11 };

auto it = std::find_first_of(v.cbegin(), v.cend(),
                             p.cbegin(), p.cend());
if (it != v.cend())
  std::cout << "found " << *it
            << " at index " << std::distance(v.cbegin(), it)
            << std::endl;
```

- Use `std::find_end()` to find the last occurrence of a subrange of elements in a range; this algorithm returns an iterator to the first element of the last subrange in the range:

```
std::vector<int> v1{ 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1 };
std::vector<int> v2{ 1, 0, 1 };

auto it = std::find_end(v1.cbegin(), v1.cend(),
                        v2.cbegin(), v2.cend());
if (it != v1.cend())
  std::cout << "found at index "
            << std::distance(v1.cbegin(), it) << std::endl;
```

- Use `std::search()` to search for the first occurrence of a subrange in a range; this

algorithm returns an iterator to the first element of the subrange in the range:

```
auto text = "The quick brown fox jumps over the lazy dog"s;
auto word = "over"s;

auto it = std::search(text.cbegin(), text.cend(),
                      word.cbegin(), word.cend());

if (it != text.cend())
  std::cout << "found " << word
            << " at index "
            << std::distance(text.cbegin(), it) << std::endl;
```

- Use `std::search()` with a *searcher*, which is a class that implements a searching algorithm and meets some predefined criteria. This overload of `std::search()` was introduced in C++17, and available standard searchers implement the *Boyer-Moore* and the *Boyer-Moore-Horspool* string searching algorithms:

```
auto text = "The quick brown fox jumps over the lazy dog"s;
auto word = "over"s;

auto it = std::search(
  text.cbegin(), text.cend(),
  std::make_boyer_moore_searcher(word.cbegin(), word.cend()));

if (it != text.cend())
  std::cout << "found " << word
            << " at index "
            << std::distance(text.cbegin(), it) << std::endl;
```

- Use `std::search_n()` to search for *N* consecutive occurrences of a value in a range; this algorithm returns an iterator to the first element of the found sequence in the range:

```
std::vector<int> v{ 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1 };

auto it = std::search_n(v.cbegin(), v.cend(), 2, 0);
if (it != v.cend())
  std::cout << "found at index "
            << std::distance(v.cbegin(), it) << std::endl;
```

- Use `std::adjacent_find()` to find two adjacent elements in a range that are equal or satisfy a binary predicate; this algorithm returns an iterator to the first element that is found:

```
std::vector<int> v{ 1, 1, 2, 3, 5, 8, 13 };

auto it = std::adjacent_find(v.cbegin(), v.cend());
if (it != v.cend())
  std::cout << "found at index "
            << std::distance(v.cbegin(), it) << std::endl;

auto it = std::adjacent_find(
  v.cbegin(), v.cend(),
  [](int const a, int const b) {
    return IsPrime(a) && IsPrime(b); });

if (it != v.cend())
  std::cout << "found at index "
            << std::distance(v.cbegin(), it) << std::endl;
```

- Use `std::binary_search()` to find whether an element exists in a sorted range; this algorithm returns a Boolean value to indicate whether the value was found or not:

```
std::vector<int> v{ 1, 1, 2, 3, 5, 8, 13 };
```

```
auto success = std::binary_search(v.cbegin(), v.cend(), 8);
if (success) std::cout << "found" << std::endl;
```

- Use `std::lower_bound()` to find the first element in a range not less than a specified value; this algorithm returns an iterator to the element:

```
std::vector<int> v{ 1, 1, 2, 3, 5, 8, 13 };

auto it = std::lower_bound(v.cbegin(), v.cend(), 1);
if (it != v.cend())
  std::cout << "lower bound at "
            << std::distance(v.cbegin(), it) << std::endl;
```

- Use `std::upper_bound()` to find the first element in a range greater than a specified value; this algorithm returns an iterator to the element:

```
std::vector<int> v{ 1, 1, 2, 3, 5, 8, 13 };

auto it = std::upper_bound(v.cbegin(), v.cend(), 1);
if (it != v.cend())
  std::cout << "upper bound at "
            << std::distance(v.cbegin(), it) << std::endl;
```

- Use `std::equal_range()` to find a subrange in a range whose values are equal to a specified value. This algorithm returns a pair of iterators defining the first and the one-past-end iterators to the subrange; these two iterators are equivalent to those returned by `std::lower_bound()` and `std::upper_bound()`:

```
std::vector<int> v{ 1, 1, 2, 3, 5, 8, 13 };

auto bounds = std::equal_range(v.cbegin(), v.cend(), 1);
std::cout << "range between indexes "
          << std::distance(v.cbegin(), bounds.first)
          << " and "
          << std::distance(v.cbegin(), bounds.second)
          << std::endl;
```

How it works…

The way these algorithms work is very similar: they all take as arguments iterators that define the searchable range and additional arguments that depend on each algorithm. Except for `std::search()`, which returns a Boolean, and `std::equal_range()`, which returns a pair of iterators, they all return an iterator to the searched element or to a subrange. These iterators must be compared with the end iterator (that is, the past-last-element) of the range to check whether the search was successful or not. If the search did not find an element or a subrange, then the returned value is the end iterator.

All these algorithms have multiple overloads, but in the *How to do it…* section, we only looked at one particular overload to show how the algorithm can be used. For a complete reference of all overloads, you should see other sources.

In all the preceding examples, we used constant iterators, but all these algorithms work the same with mutable iterators and with reverse iterators. Because they take iterators as input arguments, they can work with standard containers, C-like arrays, or anything that represents a sequence and has iterators available.

A special note on the `std::binary_search()` algorithm is necessary: the iterator parameters that define the range to search in should at least meet the requirements of the forward iterators. Regardless of the type of the supplied iterators, the number of comparisons is always logarithmic on the size of the range. However, the number of iterator increments is different if the iterators are random access, in which case the number of increments is also logarithmic, or are not random access, in which case, it is linear and proportional to the size of the range.

All these algorithms, except for `std::find_if_not()`, were available before C++11. However, some overloads of them have been introduced in the newer standards. An example is `std::search()` that has several overloads introduced in C++17. One of these overloads has the following form:

```
template<class ForwardIterator, class Searcher>
ForwardIterator search(ForwardIterator first, ForwardIterator last,
                       const Searcher& searcher );
```

This overload searches for the occurrence of a pattern defined by a searcher function object for which the standard provides several implementations:

- `default_searcher` basically delegates the searching to the standard `std::search()` algorithm.
- `boyer_moore_searcher` implements the Boyer-Moore algorithm for string searching.
- `boyer_moore_horspool_algorithm` implements the Boyer-Moore-Horspool algorithm for string searching.

There’s more…

Many standard containers have a member function `find()`, for finding elements in the container. When such a method is available and suits your needs, it should be preferred to the general algorithms because these member functions are optimized based on the particularities of each container.

See also

- *Using vector as a default container*
- *Initializing a range*
- *Using set operations on a range*
- *Sorting a range*

Sorting a range

In the previous recipe, we looked at the standard general algorithms for searching in a range. Another common operation we often need to do is sorting a range because many routines, including some of the algorithms for searching, require a sorted range. The standard library provides several general algorithms for sorting ranges, and in this recipe, we will see what these algorithms are and how they can be used.

Getting ready

The sorting general algorithms work with ranges defined by a start and end iterator and, therefore, can sort standard containers, C-like arrays, or anything that represents a sequence and has random iterators available. However, all the examples in this recipe will use `std::vector`.

How to do it…

The following is a list of standard general algorithms for searching a range:

- Use `std::sort()` for sorting a range:

```
std::vector<int> v{3, 13, 5, 8, 1, 2, 1};

std::sort(v.begin(), v.end());
// v = {1, 1, 2, 3, 5, 8, 13}

std::sort(v.begin(), v.end(), std::greater<>());
// v = {13, 8, 5, 3, 2, 1 ,1}
```

- Use `std::stable_sort()` for sorting a range but keeping the order of the equal elements:

```
struct Task
{
  int priority;
  std::string name;
};

bool operator<(Task const & lhs, Task const & rhs) {
  return lhs.priority < rhs.priority;
}

bool operator>(Task const & lhs, Task const & rhs) {
  return lhs.priority > rhs.priority;
}

std::vector<Task> v{
  { 10, "Task 1"s }, { 40, "Task 2"s }, { 25, "Task 3"s },
  { 10, "Task 4"s }, { 80, "Task 5"s }, { 10, "Task 6"s },
};

std::stable_sort(v.begin(), v.end());
// {{ 10, "Task 1" },{ 10, "Task 4" },{ 10, "Task 6" },
//  { 25, "Task 3" },{ 40, "Task 2" },{ 80, "Task 5" }}

std::stable_sort(v.begin(), v.end(), std::greater<>());
// {{ 80, "Task 5" },{ 40, "Task 2" },{ 25, "Task 3" },
//  { 10, "Task 1" },{ 10, "Task 4" },{ 10, "Task 6" }}
```

- Use `std::partial_sort()` for sorting a part of a range (and leaving the rest in an unspecified order):

```
std::vector<int> v{ 3, 13, 5, 8, 1, 2, 1 };

std::partial_sort(v.begin(), v.begin() + 4, v.end());
// v = {1, 1, 2, 3, ?, ?, ?}

std::partial_sort(v.begin(), v.begin() + 4, v.end(),
                  std::greater<>());
// v = {13, 8, 5, 3, ?, ?, ?}
```

- Use `std::partial_sort_copy()` for sorting a part of a range by copying the sorted elements to a second range and leaving the original range unchanged:

```
std::vector<int> v{ 3, 13, 5, 8, 1, 2, 1 };
std::vector<int> vc(v.size());

std::partial_sort_copy(v.begin(), v.end(),
                       vc.begin(), vc.end());
// v = {3, 13, 5, 8, 1, 2, 1}
```

```
// vc = {1, 1, 2, 3, 5, 8, 13}

std::partial_sort_copy(v.begin(), v.end(),
                       vc.begin(), vc.end(), std::greater<>());
// vc = {13, 8, 5, 3, 2, 1, 1}
```

- Use `std::nth_element()` for sorting a range so that the *N*th element is the one that would be in that position if the range was completely sorted, and the elements before it are all smaller and the ones after it are all greater, without any guarantee that they are also ordered:

```
std::vector<int> v{ 3, 13, 5, 8, 1, 2, 1 };

std::nth_element(v.begin(), v.begin() + 3, v.end());
// v = {1, 1, 2, 3, 5, 8, 13}

std::nth_element(v.begin(), v.begin() + 3, v.end(),
                 std::greater<>());
// v = {13, 8, 5, 3, 2, 1, 1}
```

- Use `std::is_sorted()` to check whether a range is sorted:

```
std::vector<int> v { 1, 1, 2, 3, 5, 8, 13 };

auto sorted = std::is_sorted(v.cbegin(), v.cend());
sorted = std::is_sorted(v.cbegin(), v.cend(),
                        std::greater<>());
```

- Use `std::is_sorted_until()` to find a sorted subrange from the beginning of a range:

```
std::vector<int> v{ 3, 13, 5, 8, 1, 2, 1 };

auto it = std::is_sorted_until(v.cbegin(), v.cend());
auto length = std::distance(v.cbegin(), it);
```

How it works…

All the preceding general algorithms take random iterators as arguments to define the range to be sorted and, some of them additionally take an output range. They all have overloads, one that requires a comparison function for sorting the elements, and one that does not and uses `operator<` for comparing the elements.

These algorithms work in the following way:

- `std::sort()` modifies the input range so that its elements are sorted according to the default or the specified comparison function; the actual algorithm for sorting is an implementation detail.
- `std::stable_sort()` is similar to `std::sort()`, but it guarantees to preserve the original order of elements that are equal.
- `std::partial_sort()` takes three iterator arguments indicating the first, middle, and last element in a range, where middle can be any element, not just the one at the natural middle position. The result is a partially sorted range so that that first `middle - first` smallest elements from the original range, that is, `[first, last)`, are found in the `[first, middle)` subrange and the rest of the elements are in an unspecified order, in the `[middle, last)` subrange.
- `std::partial_sort_copy()` is not a variant of `std::partial_copy()`, as the name may suggest, but of `std::sort()`. It sorts a range without altering it by copying its elements to an output range. The arguments of the algorithm are the first and last iterators of the input and output ranges. If the output range has a size *M* that is greater than or equal to the size *N* of the input range, the input range is entirely sorted and copied to the output range; the first *N* elements of the output range are overwritten, and the last *M* - *N* elements are left untouched. If the output range is smaller than the input range, then only the first *M* sorted elements from the input range are copied to the output range (which is entirely overwritten in this case).
- `std::nth_element()` is basically an implementation of a selection algorithm, which is an algorithm for finding the *N*th smallest element of a range. This algorithm takes three iterator arguments representing the first, *N*th, and last element, and partially sorts the range so that after sorting, the *N*th element is the one that would be in that position if the range had been entirely sorted. In the modified range, all the *N-1* elements before the *n*th one are smaller than it, and all the elements after the *n*th element are greater than it. However, there is no guarantee on the order of these other elements.
- `std::is_sorted()` checks whether the specified range is sorted according to the specified or default comparison function and returns a Boolean value to indicate that.
- `std::is_sorted_until()` finds a sorted subrange of the specified range, starting from the beginning, using either a provided comparison function or the default `operator<`. The returned value is an iterator representing the upper bound of the sorted subrange, which is also the iterator of the one-past-last sorted element.

There’s more…

Some standard containers, `std::list` and `std::forward_list`, provide a member function, `sort()`, which is optimized for those containers. These member functions should be preferred over the general standard algorithm, `std::sort()`.

See also

- *Using vector as a default container*
- *Initializing a range*
- *Using set operations on a range*
- *Finding elements in a range*

Initializing a range

In the previous recipes, we explored the general standard algorithms for searching in a range and sorting a range. The algorithms library provides many other general algorithms and among them are several that are intended for filling a range with values. In this recipe, you will learn what these algorithms are and how they should be used.

Getting ready

All the examples in this recipe use `std::vector`. However, like all the general algorithms, the ones we will see in this recipe take iterators to define the bounds of a range and can therefore be used with any standard container, C-like arrays, or custom types representing a sequence that have forward iterators defined.

Except for `std::iota()`, which is available in the `<numeric>` header, all the other algorithms are found in the `<algorithm>` header.

How to do it…

To assign values to a range, use any of the following standard algorithms:

- `std::fill()` to assign a value to all the elements of a range; the range is defined by a first and last forward iterator:

```
std::vector<int> v(5);
std::fill(v.begin(), v.end(), 42);
// v = {42, 42, 42, 42, 42}
```

- `std::fill_n()` to assign values to a number of elements of a range; the range is defined by a first forward iterator and a counter that indicates how many elements should be assigned the specified value:

```
std::vector<int> v(10);
std::fill_n(v.begin(), 5, 42);
// v = {42, 42, 42, 42, 42, 0, 0, 0, 0, 0}
```

- `std::generate()` to assign the value returned by a function to the elements of a range; the range is defined by a first and last forward iterator, and the function is invoked once for each element in the range:

```
std::random_device rd{};
std::mt19937 mt{ rd() };
std::uniform_int_distribution<> ud{1, 10};
std::vector<int> v(5);
std::generate(v.begin(), v.end(),
              [&ud, &mt] {return ud(mt); });
```

- `std::generate_n()` to assign the value returned by a function to a number of elements of a range; the range is defined by a first forward iterator and a counter that indicates how many elements should be assigned the value from the function that is invoked once for each element:

```
std::vector<int> v(5);
auto i = 1;
std::generate_n(v.begin(), v.size(), [&i] { return i*i++; });
// v = {1, 4, 9, 16, 25}
```

- `std::iota()` to assign sequentially increasing values to the elements of a range; the range is defined by a first and last forward iterator, and the values are incremented using the prefix `operator++` from an initial specified value:

```
std::vector<int> v(5);
std::iota(v.begin(), v.end(), 1);
// v = {1, 2, 3, 4, 5}
```

How it works…

`std::fill()` and `std::fill_n()` work similarly but differ in the way the range is specified: for the former by a first and last iterator, for the latter by a first iterator and a count. The second algorithm returns an iterator, representing either the one-past-last assigned element if the counter is greater than zero, or an iterator to the first element of the range otherwise.

`std::generate()` and `std::generate_n()` are also similar, differing only in the way the range is specified. The first takes two iterators, defining the range's lower and upper bounds, and the second, an iterator to the first element and a count. Like `std::fill_n()`, `std::generate_n()` also returns an iterator, representing either the one-past-last assigned element if the count is greater than zero, or an iterator to the first element of the range, otherwise. These algorithms call a specified function for each element in the range and assign the returned value to the element. The generating function does not take any argument, so the value of the argument cannot be passed to the function as this is intended as a function to initialize the elements of a range. If you need to use the value of the elements to generate new values, you should use `std::transform()`.

`std::iota()` takes its name from the ι (iota) function from the APL programming language, and though it was a part of the initial STL, it was only included in the standard library in C++11. This function takes a first and last iterator to a range and an initial value that is assigned to the first element of the range and then used to generate sequentially increasing values using the prefix `operator++` for the rest of the elements in the range.

See also

- *Using vector as a default container*
- *Sorting a range*
- *Using set operations on a range*
- *Finding elements in a range*
- *Generating pseudo-random numbers* recipe of Chapter 2, *Working with Numbers and Strings*
- *Initializing all bits of internal state of a pseudo-random number generator* recipe of Chapter 2, *Working with Numbers and Strings*

Using set operations on a range

The standard library provides several algorithms for set operations that enable us to do unions, intersections, or differences of sorted ranges. In this recipe, we will see what these algorithms are and how they work.

Getting ready

The algorithms for set operations work with iterators, which means they can be used for standard containers, C-like arrays, or any custom type representing a sequence that has input iterators available. All the examples in this recipe will use `std::vector`.

For all the examples in the next section, we will use the following ranges:

```
std::vector<int> v1{ 1, 2, 3, 4, 4, 5 };
std::vector<int> v2{ 2, 3, 3, 4, 6, 8 };
std::vector<int> v3;
```

How to do it…

Use the following general algorithms for set operations:

- `std::set_union()` to compute the union of two ranges into a third range:

```
std::set_union(v1.cbegin(), v1.cend(),
               v2.cbegin(), v2.cend(),
               std::back_inserter(v3));
// v3 = {1, 2, 3, 3, 4, 4, 5, 6, 8}
```

- `std::merge()` to merge the content of two ranges into a third one; this is similar to `std::set_union()` except that it copies the entire content of the input ranges into the output one, not just their union:

```
std::merge(v1.cbegin(), v1.cend(),
           v2.cbegin(), v2.cend(),
           std::back_inserter(v3));
// v3 = {1, 2, 2, 3, 3, 3, 4, 4, 4, 5, 6, 8}
```

- `std::set_intersection()` to compute the intersection of the two ranges into a third range:

```
std::set_intersection(v1.cbegin(), v1.cend(),
                      v2.cbegin(), v2.cend(),
                      std::back_inserter(v3));
// v3 = {2, 3, 4}
```

- `std::set_difference()` to compute the difference of two ranges into a third range; the output range will contain elements from the first range, which are not present in the second range:

```
std::set_difference(v1.cbegin(), v1.cend(),
                    v2.cbegin(), v2.cend(),
                    std::back_inserter(v3));
// v3 = {1, 4, 5}
```

- `std::set_symmetric_difference()` to compute a dual difference of the two ranges into a third range; the output range will contain elements that are present in any of the input ranges, but only in one:

```
std::set_symmetric_difference(v1.cbegin(), v1.cend(),
                              v2.cbegin(), v2.cend(),
                              std::back_inserter(v3));
// v3 = {1, 3, 4, 5, 6, 8}
```

- `std::includes()` to check if one range is a subset of another range (that is, all its elements are also present in the other range):

```
std::vector<int> v1{ 1, 2, 3, 4, 4, 5 };
std::vector<int> v2{ 2, 3, 3, 4, 6, 8 };
std::vector<int> v3{ 1, 2, 4 };
std::vector<int> v4{ };

auto i1 = std::includes(v1.cbegin(), v1.cend(),
                        v2.cbegin(), v2.cend()); // i1 = false
auto i2 = std::includes(v1.cbegin(), v1.cend(),
                        v3.cbegin(), v3.cend()); // i2 = true
auto i3 = std::includes(v1.cbegin(), v1.cend(),
```

```
                                v4.cbegin(), v4.cend()); // i3 = true
```

How it works…

All the set operations that produce a new range from two input ranges, in fact, have the same interface and work in a similar way:

- They take two input ranges, each defined by a first and last input iterator.
- They take an output iterator to an output range where elements will be inserted.
- They have an overload that takes an extra argument representing a comparison binary function object that must return `true` if the first argument is less than the second. When a comparison function object is not specified, `operator<` is used.
- They return an iterator past the end of the constructed output range.
- The input ranges must be sorted using either `operator<` or the provided comparison function, depending on the overload that is used.
- The output range must not overlap any of the two input ranges.

We will demonstrate the way they work with additional examples using vectors of a POD type `Task` that we also used in a previous recipe:

```
struct Task
{
  int priority;
  std::string name;
};

bool operator<(Task const & lhs, Task const & rhs) {
  return lhs.priority < rhs.priority;
}

bool operator>(Task const & lhs, Task const & rhs) {
  return lhs.priority > rhs.priority;
}

std::vector<Task> v1{
  { 10, "Task 1.1"s },
  { 20, "Task 1.2"s },
  { 20, "Task 1.3"s },
  { 20, "Task 1.4"s },
  { 30, "Task 1.5"s },
  { 50, "Task 1.6"s },
};

std::vector<Task> v2{
  { 20, "Task 2.1"s },
  { 30, "Task 2.2"s },
  { 30, "Task 2.3"s },
  { 30, "Task 2.4"s },
  { 40, "Task 2.5"s },
  { 50, "Task 2.6"s },
};
```

The particular way each algorithm produces the output range is described here:

- `std::set_union()` copies all the elements present in one or both of the input ranges to the output range, producing a new sorted range. If an element is found *M* times in the first range and *N* times in the second range, then all the *M* elements from the first range will be copied to the output range in their existing order, and then the *N*-*M* elements from the second range are copied to the output range if $N > M$, or

0 elements otherwise:

```
std::vector<Task> v3;
std::set_union(v1.cbegin(), v1.cend(),
               v2.cbegin(), v2.cend(),
               std::back_inserter(v3));
// v3 = {{10, "Task 1.1"},{20, "Task 1.2"},{20, "Task 1.3"},
//       {20, "Task 1.4"},{30, "Task 1.5"},{30, "Task 2.3"},
//       {30, "Task 2.4"},{40, "Task 2.5"},{50, "Task 1.6"}}
```

- `std::merge()` copies all the elements from both the input ranges into the output range, producing a new range sorted with respect to the comparison function:

```
std::vector<Task> v4;
std::merge(v1.cbegin(), v1.cend(),
           v2.cbegin(), v2.cend(),
           std::back_inserter(v4));
// v4 = {{10, "Task 1.1"},{20, "Task 1.2"},{20, "Task 1.3"},
//       {20, "Task 1.4"},{20, "Task 2.1"},{30, "Task 1.5"},
//       {30, "Task 2.2"},{30, "Task 2.3"},{30, "Task 2.4"},
//       {40, "Task 2.5"},{50, "Task 1.6"},{50, "Task 2.6"}}
```

- `std::set_intersection()` copies all the elements that are found in both the input ranges into the output range, producing a new range sorted with respect to the comparison function:

```
std::vector<Task> v5;
std::set_intersection(v1.cbegin(), v1.cend(),
                      v2.cbegin(), v2.cend(),
                      std::back_inserter(v5));
// v5 = {{20, "Task 1.2"},{30, "Task 1.5"},{50, "Task 1.6"}}
```

- `std::set_difference()` copies to the output range all the elements from the first input range that are not found in the second input range. For equivalent elements that are found in both the ranges, the following rule applies: if an element is found *M* times in the first range and *N* times in the second range, and if *M* > *N*, then it is copied *M*-*N* times; otherwise it is not copied:

```
std::vector<Task> v6;
std::set_difference(v1.cbegin(), v1.cend(),
                    v2.cbegin(), v2.cend(),
                    std::back_inserter(v6));
// v6 = {{10, "Task 1.1"},{20, "Task 1.3"},{20, "Task 1.4"}}
```

- `std::set_symmetric_difference()` copies to the output range all the elements that are found in either of the two input ranges but not in both of them. If an element is found *M* times in the first range and *N* times in the second range, then if *M* > *N*, the last *M*-*N* of those elements from the first range are copied into the output rage, else, the last *N*-*M* of those elements from the second range will be copied into the output range:

```
std::vector<Task> v7;
std::set_symmetric_difference(v1.cbegin(), v1.cend(),
                              v2.cbegin(), v2.cend(),
                              std::back_inserter(v7));
// v7 = {{10, "Task 1.1"},{20, "Task 1.3"},{20, "Task 1.4"}
//       {30, "Task 2.3"},{30, "Task 2.4"},{40, "Task 2.5"}}
```

On the other hand, `std::includes()` does not produce an output range; it only checks whether the second range is included in the first range. It returns a Boolean value that is

`true` if the second range is empty or all its elements are included in the first range, or `false` otherwise. It also has two overloads, one of them specifying a comparison binary function object.

See also

- *Using vector as a default container*
- *Sorting a range*
- *Initializing a range*
- *Using iterators to insert new elements in a container*
- *Finding elements in a range*

Using iterators to insert new elements in a container

When you’re working with containers, it is often useful to insert new elements at the beginning, end, or somewhere in the middle. There are algorithms, such as the ones we saw in the previous recipe, *Using set operations on a range*, that require an iterator to a range to insert into, but if you simply pass an iterator, such as the one returned by `begin()`, it will not insert but overwrite the elements of the container. Moreover, it’s not possible to insert at the end by using the iterator returned by `end()`. In order to perform such operations, the standard library provides a set of iterators and iterator adapters that enable these scenarios.

Getting ready

The iterators and adapters discussed in this recipe are available in the `std` namespace in the `<iterator>` header. If you include headers such as, `<algorithm>`, you do not have to explicitly include `<iterator>`.

How to do it…

Use the following iterator adapters to insert new elements in a container:

- `std::back_inserter()` to insert elements at the end, for containers that have a `push_back()` method:

```
std::vector<int> v{ 1,2,3,4,5 };
std::fill_n(std::back_inserter(v), 3, 0);
// v={1,2,3,4,5,0,0,0}
```

- `std::front_inserter()` to insert elements at the beginning, for containers that have a `push_front()` method:

```
std::list<int> l{ 1,2,3,4,5 };
std::fill_n(std::front_inserter(l), 3, 0);
// l={0,0,0,1,2,3,4,5}
```

- `std::inserter()` to insert anywhere in a container, for containers that have an `insert()` method:

```
std::vector<int> v{ 1,2,3,4,5 };
std::fill_n(std::inserter(v, v.begin()), 3, 0);
// v={0,0,0,1,2,3,4,5}

std::list<int> l{ 1,2,3,4,5 };
auto it = l.begin();
std::advance(it, 3);
std::fill_n(std::inserter(l, it), 3, 0);
// l={1,2,3,0,0,0,4,5}
```

How it works…

`std::back_inserter()`, `std::front_inserter()`, and `std::inserter()` are all helper functions that create iterator adapters of types, `std::back_insert_iterator`, `std::front_insert_iterator`, and `std::insert_iterator`. These are all output iterators that append, prepend, or insert into the container for which they were constructed. Incrementing and dereferencing these iterators does not do anything. However, upon assignment, these iterators call the following methods from the container:

- `std::back_insterter_iterator` calls `push_back()`
- `std::front_inserter_iterator` calls `push_front()`
- `std::insert_iterator` calls `insert()`

The following is the over-simplified implementation of `std::back_inserter_iterator`:

```
template<class C>
class back_insert_iterator {
public:
  typedef back_insert_iterator<C> T;
  typedef typename C::value_type V;

  explicit back_insert_iterator( C& c ) :container( &c ) { }

  T& operator=( const V& val ) {
    container->push_back( val );
    return *this;
  }

  T& operator*() { return *this; }

  T& operator++() { return *this; }

  T& operator++( int ) { return *this; }
  protected:
  C* container;
};
```

Because of the way the assignment operator works, these iterators can only be used with some standard containers:

- `std::back_insert_iterator` can be used with `std::vector`, `std::list`, `std::deque`, and `std::basic_string`.
- `std::front_insert_iterator` can be used with `std::list`, `std::forward_list`, and `std:deque`.
- `std::insert_iterator` can be used with all the standard containers.

The following example inserts three elements with the value 0 at the beginning of an `std::vector`:

```
std::vector<int> v{ 1,2,3,4,5 };
std::fill_n(std::inserter(v, v.begin()), 3, 0);
// v={0,0,0,1,2,3,4,5}
```

The `std::inserter()` adapter takes two arguments: the container, and the iterator where an element is supposed to be inserted. Upon calling `insert()` on the container, the `std::insert_iterator` increments the iterator, so upon being assigned again, it can insert a new element into the next position. Here is how the assignment operator is implemented

for this iterator adapter:

```
    T& operator=(const V& v)
    {
      iter = container->insert(iter, v);
      ++iter;
      return (*this);
    }
```

There’s more…

These iterator adapters are intended to be used with algorithms or functions that insert multiple elements into a range. They can be used, of course, to insert a single element, but that is rather an anti-pattern, since simply calling `push_back()`, `push_front()`, or `insert()` is much simpler and intuitive in this case. The following examples should be avoided:

```
std::vector<int> v{ 1,2,3,4,5 };
*std::back_inserter(v) = 6; // v = {1,2,3,4,5,6}

std::back_insert_iterator<std::vector<int>> it(v);
*it = 7;                    // v = {1,2,3,4,5,6,7}
```

See also

- *Using set operations on a range*

Writing your own random access iterator

In the first chapter, we saw how we can enable range-based for loops for custom types by implementing iterators and free `begin()` and `end()` functions to return iterators to the first and one-past-the-last element of the custom range. You might have noticed that the minimal iterator implementation that we provided in that recipe does not meet the requirements for a standard iterator because it cannot be copy constructible or assigned and cannot be incremented. In this recipe, we will build upon that example and show how to create a random access iterator that meets all requirements.

Getting ready

For this recipe, you should know the types of iterators the standard defines and how they are different. A good overview of their requirements is available at http://www.cplusplus.com/reference/iterator/.

To exemplify how to write a random access iterator, we will consider a variant of the `dummy_array` class used in the *Enabling range-based for loops for custom types* recipe of Chapter 1, *Learning Modern Core Language Features*. This is a very simple array concept, with no practical value, other than serving as a code base for demonstrating iterators:

```
template <typename Type, size_t const SIZE>
class dummy_array
{
  Type data[SIZE] = {};
public:
  Type& operator[](size_t const index)
  {
    if (index < SIZE) return data[index];
    throw std::out_of_range("index out of range");
  }

 Type const & operator[](size_t const index) const
 {
   if (index < SIZE) return data[index];
   throw std::out_of_range("index out of range");
 }

  size_t size() const { return SIZE; }
};
```

All the code shown in the next section, the iterator classes, `typedef`s, and the `begin()` and `end()` functions, will be a part of this class.

How to do it…

To provide mutable and constant random access iterators for the `dummy_array` class shown in the previous section, add the following members to the class:

- An iterator class template, which is parameterized with the type of elements and the size of the array. The class must have the following public `typedef`s that define standard synonyms:

```
template <typename T, size_t const Size>
class dummy_array_iterator
{
public:
  typedef dummy_array_iterator            self_type;
  typedef T                               value_type;
  typedef T&                              reference;
  typedef T*                              pointer;
  typedef std::random_access_iterator_tag iterator_category;
  typedef ptrdiff_t                       difference_type;
};
```

- Private members for the iterator class: a pointer to the array data and a current index into the array:

```
private:
   pointer ptr = nullptr;
   size_t index = 0;
```

- Private method for the iterator class to check whether two iterator instances point to the same array data:

```
private:
  bool compatible(self_type const & other) const
  {
    return ptr == other.ptr;
  }
```

- An explicit constructor for the iterator class:

```
public:
   explicit dummy_array_iterator(pointer ptr,
                                 size_t const index)
     : ptr(ptr), index(index) { }
```

- Iterator class members to meet common requirements for all iterators: copy-constructible, copy-assignable, destructible, prefix, and postfix incrementable. In this implementation, the post increment operator is implemented in terms of the pre-increment operator to avoid code duplication:

```
dummy_array_iterator(dummy_array_iterator const & o)
   = default;
dummy_array_iterator& operator=(dummy_array_iterator const & o)
   = default;
~dummy_array_iterator() = default;

self_type & operator++ ()
{
   if (index >= Size)
```

```
      throw std::out_of_range("Iterator cannot be incremented past
                               the end of range.");
  ++index;
  return *this;
}

self_type operator++ (int)
{
  self_type tmp = *this;
  ++*this;
  return tmp;
}
```

- Iterator class members to meet input iterator requirements: test for equality/inequality, dereferenceable as rvalues:

```
bool operator== (self_type const & other) const
{
  assert(compatible(other));
  return index == other.index;
}

bool operator!= (self_type const & other) const
{
  return !(*this == other);
}

reference operator* () const
{
  if (ptr == nullptr)
    throw std::bad_function_call();
  return *(ptr + index);
}

reference operator-> () const
{
  if (ptr == nullptr)
    throw std::bad_function_call();
  return *(ptr + index);
}
```

- Iterator class members to meet forward iterator requirements: default constructible:

```
dummy_array_iterator() = default;
```

- Iterator class members to meet bidirectional iterator requirements: decrementable:

```
self_type & operator--()
{
  if (index <= 0)
    throw std::out_of_range("Iterator cannot be decremented
                             past the end of range.");
  --index;
  return *this;
}

self_type operator--(int)
{
  self_type tmp = *this;
  --*this;
  return tmp;
}
```

- Iterator class members to meet random access iterator requirements: arithmetic add and subtract, comparable for inequality with other iterators, compound assignments, and offset dereferenceable:

```
self_type operator+(difference_type offset) const
```

```
  {
    self_type tmp = *this;
    return tmp += offset;
  }

  self_type operator-(difference_type offset) const
  {
    self_type tmp = *this;
    return tmp -= offset;
  }

  difference_type operator-(self_type const & other) const
  {
    assert(compatible(other));
    return (index - other.index);
  }

  bool operator<(self_type const & other) const
  {
    assert(compatible(other));
    return index < other.index;
  }

  bool operator>(self_type const & other) const
  {
    return other < *this;
  }

  bool operator<=(self_type const & other) const
  {
    return !(other < *this);
  }

  bool operator>=(self_type const & other) const
  {
    return !(*this < other);
  }

  self_type & operator+=(difference_type const offset)
  {
    if (index + offset < 0 || index + offset > Size)
      throw std::out_of_range("Iterator cannot be incremented
                               past the end of range.");
    index += offset;
    return *this;
  }

  self_type & operator-=(difference_type const offset)
  {
    return *this += -offset;
  }

  value_type & operator[](difference_type const offset)
  {
    return (*(*this + offset));
  }

  value_type const & operator[](difference_type const offset) const
  {
    return (*(*this + offset));
  }
```

- Add `typedef`s to the `dummy_array` class for mutable and constant iterator synonyms:

```
  public:
     typedef dummy_array_iterator<Type, SIZE>
             iterator;
     typedef dummy_array_iterator<Type const, SIZE>
             constant_iterator;
```

- Add the public `begin()` and `end()` functions to the `dummy_array` class to return the iterators to the first and one-past-last elements in the array:

```
        iterator begin()
        {
          return iterator(data, 0);
        }

        iterator end()
        {
          return iterator(data, SIZE);
        }

        constant_iterator begin() const
        {
          return constant_iterator(data, 0);
        }

        constant_iterator end() const
        {
          return constant_iterator(data, SIZE);
        }
```

How it works…

The standard library defines five categories of iterators:

- *Input iterators*: These are the simplest category and guarantee validity only for single-pass sequential algorithms. After being incremented, the previous copies may become invalid.
- *Output iterators*: These are basically input iterators that can be used to write to the pointed element.
- *Forward iterators*: These can read (and write) data to the pointed element. They satisfy the requirements for input iterators and, in addition, must be default constructible and must support multi-pass scenarios without invalidating the previous copies.
- *Bidirectional iterators*: These are forward iterators that, in addition, support decrementing, so they can move in both directions.
- *Random access iterators*: These support access to any element in the container in constant time. They implement all the requirements for bidirectional iterators, and, in addition, support arithmetic operations `+` and `-`, compound assignments `+=` and `-=`, comparisons with other iterators with `<`, `<=`, `>`, `>=`, and the offset dereference operator.

Forward, bidirectional, and random access iterators that also implement the requirements of output iterators are called *mutable iterators*.

In the previous section, we saw how to implement random access iterators, with a step-by-step walkthrough of the requirements of each category of iterators (as each iterator category includes the requirements of the previous category and adds new requirements). The iterator class template is common for both constant and mutable iterators, and we have defined two synonyms for it called `iterator` and `constant_iterator`.

After implementing the inner iterator class template, we also defined the `begin()` and `end()` member functions that return an iterator to the first and the one-past-last element in the array. These methods have overloads to return mutable or constant iterators, depending on whether the `dummy_array` class instance is mutable or constant.

With this implementation of the `dummy_array` class and its iterators, we can write the following samples. For more examples, check the source code that accompanies this book:

```
dummy_array<int, 3> a;
a[0] = 10;
a[1] = 20;
a[2] = 30;

std::transform(a.begin(), a.end(), a.begin(),
               [](int const e) {return e * 2; });

for (auto&& e : a) std::cout << e << std::endl;

auto lp = [](dummy_array<int, 3> const & ca)
{
  for (auto const & e : ca)
    std::cout << e << std::endl;
};
```

```
    lp(a);

    dummy_array<std::unique_ptr<Tag>, 3> ta;
    ta[0] = std::make_unique<Tag>(1, "Tag 1");
    ta[1] = std::make_unique<Tag>(2, "Tag 2");
    ta[2] = std::make_unique<Tag>(3, "Tag 3");

    for (auto it = ta.begin(); it != ta.end(); ++it)
      std::cout << it->id << " " << it->name << std::endl;
```

There’s more…

Apart from `begin()` and `end()`, a container may have additional methods such as `cbegin()`/`cend()` (for constant iterators), `rbegin()`/`rend()` (for mutable reverse iterators), and `crbegin()`/ `crend()` (for constant reverse iterators). Implementing this is left as an exercise for you.

On the other hand, in modern C++, these functions that return the first and last iterators do not have to be member functions but can be provided as non-member functions. In fact, this is the topic of the next recipe, *Container access with non-member functions*.

See also

- *Enabling range-based for loops for custom types* recipe of Chapter 1, *Learning Modern Core Language Features*
- *Creating type aliases and alias templates* recipe of Chapter 1, *Learning Modern Core Language Features*

Container access with non-member functions

Standard containers provide the `begin()` and `end()` member functions for retrieving iterators to the first and one-past-last element of the container. There are actually four sets of these functions. Apart from `begin()`/`end()`, containers provide `cbegin()`/`cend()` to return constant iterators, `rbegin()`/`rend()` to return mutable reverse iterators, and `crbegin()`/`crend()` to return constant reverse iterators. In C++11/C++14, all these have non-member equivalents that work with standard containers, C-like arrays, and any custom type that specializes them. In C++17, even more non-member functions have been added; `std::data()`—that returns a pointer to the block of memory containing the elements of the container, `std::size()`—that returns the size of a container or array, and `std::empty()`—that returns whether the given container is empty. These non-member functions are intended for generic code but can be used anywhere in your code.

Getting ready

In this recipe, we will use as an example, the `dummy_array` class and its iterators that we implemented in the previous recipe, *Writing your own random access iterator*. You should read that recipe before continuing with this one.

Non-member `begin()`/`end()` functions and the other variants, as well as non-member `data()`, `size()` and `empty()` are available in the `std` namespace in the `<iterator>` header, which is implicitly included with any of the following headers: `<array>`, `<deque>`, `<forward_list>`, `<list>`, `<map>`, `<regex>`, `<set>`, `<string>`, `<unordered_map>`, `<unordered_set>`, and `<vector>`.

In this recipe, we will refer to the `std::begin()`/`std::end()` functions, but everything discussed also applies to the other functions: `std::cbegin()`/`std::cend()`, `std::rbegin()`/`std::rend()`, and `std::crbegin()`/`std::crend()`.

How to do it…

Use the non-member `std::begin()`/`std::end()` function and the other variants, as well as `std::data()`, `std::size()` and `std::empty()` with:

- Standard containers:

```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
auto sv1 = std::size(v1);  // sv1 = 5
auto ev1 = std::empty(v1); // ev1 = false
auto dv1 = std::data(v1);  // dv1 = v1.data()
for (auto i = std::begin(v1); i != std::end(v1); ++i)
  std::cout << *i << std::endl;

std::vector<int> v2;
std::copy(std::cbegin(v1), std::cend(v1),
          std::back_inserter(v2));
```

- (C-like) arrays:

```
int a[5] = { 1, 2, 3, 4, 5 };
auto pos = std::find_if(std::crbegin(a), std::crend(a),
                        [](int const n) {return n % 2 == 0; });
auto sa = std::size(a);  // sa = 5
auto ea = std::empty(a); // ea = false
auto da = std::data(a);  // da = a
```

- Custom types that provide corresponding member functions, `begin()`/`end()`, `data()`, `empty()`, or `size()`:

```
dummy_array<std::string, 5> sa;
dummy_array<int, 5> sb;
sa[0] = "1"s;
sa[1] = "2"s;
sa[2] = "3"s;
sa[3] = "4"s;
sa[4] = "5"s;

std::transform(
  std::begin(sa), std::end(sa),
  std::begin(sb),
  [](std::string const & s) {return std::stoi(s); });
// sb = [1, 2, 3, 4, 5]

auto sa_size = std::size(sa); // sa_size = 5
```

- Generic code where the type of the container is not known:

```
template <typename F, typename C>
void process(F&& f, C const & c)
{
  std::for_each(std::begin(c), std::end(c),
                std::forward<F>(f));
}

auto l = [](auto const e) {std::cout << e << std::endl; };

process(l, v1); // std::vector<int>
process(l, a);  // int[5]
process(l, sa); // dummy_array<std::string, 5>
```

How it works…

These non-member functions were introduced in different versions of the standard, but all of them were modified in C++17 to return `constexpr auto`:

- `std::begin()` and `std::end()` in C++11
- `std::cbegin()`/`std::cend()`, `std::rbegin()`/`std::rend()`, and `std::crbegin()`/`std::crend()` in C++14
- `std::data()`, `std::size()`, and `std::empty()` in C++17

The `begin()`/`end()` family of functions have overloads for container classes and arrays, and all they do is the following:

- Return the results of calling the container-corresponding member function for containers.
- Return a pointer to the first or one-past-last element of the array for arrays.

The actual typical implementation for `std::begin()`/`std::end()` is the following:

```
template<class C>
constexpr auto inline begin(C& c) -> decltype(c.begin())
{
  return c.begin();
}
```

```
template<class C>
constexpr auto inline end(C& c) -> decltype(c.end())
{
  return c.end();
}

template<class T, std::size_t N>
constexpr T* inline begin(T (&array)[N])
{
  return array;
}

template<class T, std::size_t N>
constexpr T* inline begin(T (&array)[N])
{
  return array+N;
}
```

Custom specialization can be provided for containers that do not have corresponding `begin()`/`end()` members but can still be iterated. The standard library actually provides such specializations for `std::initializer_list` and `std::valarray`.

Specializations must be defined in the same namespace where the original class or function template has been defined. Therefore, if you want to specialize any of the `std::begin()`/`std::end()` pairs you must do it in the `std` namespace.

The other non-member functions for container access, that were introduced in C++17, have also several overloads:

- `std::data()` has several overloads; for a class `c` it returns `c.data()`, for arrays it returns the `array`, and for `std::initializer_list<T>` it returns the `il.begin()`.

```
template <class C>
constexpr auto data(C& c) -> decltype(c.data())
{
  return c.data();
}

template <class C>
constexpr auto data(const C& c) -> decltype(c.data())
{
  return c.data();
}

template <class T, std::size_t N>
constexpr T* data(T (&array)[N]) noexcept
{
  return array;
}

template <class E>
constexpr const E* data(std::initializer_list<E> il) noexcept
{
  return il.begin();
}
```

- `std::size()` has two overloads; for a class `C` it returns `c.size()`, and for arrays it returns the size `N`.

```
template <class C>
constexpr auto size(const C& c) -> decltype(c.size())
{
  return c.size();
}

template <class T, std::size_t N>
constexpr std::size_t size(const T (&array)[N]) noexcept
{
  return N;
}
```

- `std::empty()` has several overloads; for a class `C` it returns `c.empty()`, for arrays it returns `false`, and for `std::initializer_list<T>` it returns `il.size() == 0`.

```
template <class C>
constexpr auto empty(const C& c) -> decltype(c.empty())
{
  return c.empty();
}

template <class T, std::size_t N>
constexpr bool empty(const T (&array)[N]) noexcept
{
  return false;
}
```

```
template <class E>
constexpr bool empty(std::initializer_list<E> il) noexcept
{
  return il.size() == 0;
}
```

There’s more…

These non-member functions are mainly intended for template code where the container is not known and can be a standard container, a C-like array, or a custom type. Using the non-member version of these functions enables us to write simpler and less code that works with all these types of containers.

However, the use of these functions is not and should not be limited to generic code. Though it is rather a matter of personal preference, it can be a good habit to be consistent and use them everywhere in your code. All these methods have lightweight implementations that will most likely be inlined by the compiler, which means that there will be no overhead at all over using the corresponding member functions.

See also

- *Writing your own random access iterator*

General Purpose Utilities

The recipes included in this chapter are as follows:

- Expressing time intervals with chrono::duration
- Measuring function execution time with a standard clock
- Generating hash values for custom types
- Using std::any to store any value
- Using std::optional to store optional values
- Using std::variant as a type-safe union
- Visiting a std::variant
- Registering a function to be called when a program exits normally
- Using type traits to query properties of types
- Writing your own type traits
- Using std::conditional to choose between types

Introduction

The standard library contains many general purpose utilities and libraries beyond the containers, algorithms, and iterators discussed in the previous chapter. This chapter is focused on three areas: the `chrono` library for working with dates and times, type traits that provide meta-information about other times, and the new C++17 types `std::any`, `std::optional`, and `std::variant`.

Expressing time intervals with chrono::duration

Working with times and dates is a common operation regardless of the programming language. C++11 provides a flexible date and time library as part of the standard library that enables us to define time points and time intervals. This library, called `chrono`, is a general purpose utility library designed to work with a timer and clocks that can be different on different systems and, therefore, be precision-neutral. The library is available in the `<chrono>` header in the `std::chrono` namespace and defines and implements several components, as follows:

- *Durations* that represent time intervals.
- *Time points* that present a duration of time since the epoch of a clock.
- *Clocks* that define an epoch (that is, start of time) and a tick.

In this recipe, we will see how to work with durations.

Getting ready

This recipe is not intended as a complete reference to the `duration` class. It is recommended that you consult additional resources for that purpose (the library reference documentation is available at http://en.cppreference.com/w/cpp/chrono).

In the `chrono` library, a time interval is represented by a `std::chrono::duration` class.

How to do it…

To work with time intervals, use the following:

- `std::chrono::duration` typedefs for hours, minutes, seconds, milliseconds, microseconds, and nanoseconds:

```
std::chrono::hours        half_day(12);
std::chrono::minutes      half_hour(30);
std::chrono::seconds      half_minute(30);
std::chrono::milliseconds half_second(500);
std::chrono::microseconds half_millisecond(500);
std::chrono::nanoseconds  half_microsecond(500);
```

- Use the standard user-defined literal operators from C++14, available in the namespace `std::chrono_literals` for creating durations of hours, minutes, seconds, milliseconds, microseconds, and nanoseconds:

```
using namespace std::chrono_literals;

auto half_day         = 12h;
auto half_hour        = 30min;
auto half_minute      = 30s;
auto half_second      = 500ms;
auto half_millisecond = 500us;
auto half_microsecond = 500ns;
```

- Use direct conversion from a lower precision duration to a higher precision duration:

```
std::chrono::hours half_day_in_h(12);
std::chrono::minutes half_day_in_min(half_day_in_h);
std::cout << half_day_in_h.count() << "h" << std::endl;    //12h
std::cout << half_day_in_min.count() << "min" << std::endl;//720min
```

- Use `std::chrono::duration_cast` to convert from a higher precision to a lower precision duration:

```
using namespace std::chrono_literals;

auto total_seconds = 12345s;
auto hours =
   std::chrono::duration_cast<std::chrono::hours>
      (total_seconds);
auto minutes =
   std::chrono::duration_cast<std::chrono::minutes>
      (total_seconds % 1h);
auto seconds =
   std::chrono::duration_cast<std::chrono::seconds>
      (total_seconds % 1min);

std::cout << hours.count() << ':'
          << minutes.count() << ':'
          << seconds.count() << std::endl; // 3:25:45
```

- Use the conversion functions `floor()`, `round()`, and `ceil()` available in C++17 when rounding is necessary:

```
using namespace std::chrono_literals;

auto total_seconds = 12345s;
auto m1 = std::chrono::floor<std::chrono::minutes>(total_seconds);
```

```
// 205 min
auto m2 = std::chrono::round<std::chrono::minutes>(total_seconds);
// 206 min
auto m3 = std::chrono::ceil<std::chrono::minutes>(total_seconds);
// 206 min
auto sa = std::chrono::abs(total_seconds);
```

- Use arithmetic operations, compound assignments, and comparison operations to modify and compare time intervals:

```
using namespace std::chrono_literals;

auto d1 = 1h + 23min + 45s; // d1 = 5025s
auto d2 = 3h + 12min + 50s; // d2 = 11570s
if (d1 < d2) { /* do something */ }
```

How it works…

The `std::chrono::duration` class defines a number of ticks (the increment between two moments in time) over a unit of time. The default unit is the second, and for expressing other units, such as minutes or milliseconds, we need to use a ratio. For units greater than the second, the ratio is greater than one, such as `ratio<60>` for minutes. For units smaller than the second, the ratio is smaller than one, such as `ratio<1, 1000>` for milliseconds. The number of ticks can be retrieved with the `count()` member function.

The standard library defines several type synonyms for durations of nanoseconds, microseconds, milliseconds, seconds, minutes, and hours that we used in the first example in the previous section. The following code shows how these durations are defined in the `chrono` namespace:

```
namespace std {
  namespace chrono {
    typedef duration<long long, ratio<1, 1000000000>> nanoseconds;
    typedef duration<long long, ratio<1, 1000000>> microseconds;
    typedef duration<long long, ratio<1, 1000>> milliseconds;
    typedef duration<long long> seconds;
    typedef duration<int, ratio<60> > minutes;
    typedef duration<int, ratio<3600> > hours;
  }
}
```

However, with this flexible definition, we can express time intervals such as *1.2 sixths of a minute* (which means 12 seconds), where 1.2 is the number of ticks of the duration and `ratio<10>` (as in 60/6) is the time unit:

```
std::chrono::duration<double, std::ratio<10>> d(1.2); // 12 sec
```

In C++14, several standard user-defined literal operators have been added to the namespace `std::chrono_literals`. This makes it easier to define durations, but you must include the namespace in the scope where you want to use the literal operators.

You should only include namespaces for user-defined literal operators in the scope where you want to use them, and not in larger scopes, in order to avoid conflict with other operators with the same name from different libraries and namespaces.

All arithmetic operations are available for the `duration` class. It is possible to add and subtract durations, multiply or divide them by a value, or apply the `modulo` operation. However, it is important to note that when two durations of different time units are added or subtracted, the result is a duration of the greatest common divisor of the two time units. That means that if you add a duration representing seconds and a duration representing minutes, the result is a duration representing seconds.

Conversion from a duration with a less precise time unit to a duration with a more precise time unit is done implicitly. On the other hand, conversion from a more precise to a less precise time unit requires an explicit cast. This is done with the non-member function `std::chrono::duration_cast()`. In the *How to do it…* section, we have seen an example for determining the number of hours, minutes, and seconds of a given duration expressed in

seconds.

C++17 has added several more non-member conversion functions that perform duration casting with rounding: `floor()` to round down, `ceil()` to round up, and `round()` to round to the nearest. Also, C++17 added a non-member function `abs()` to retain the absolute value of a duration.

There’s more…

`chrono` is a general purpose library, and because of that, it lacks many useful particular features, such as expressing a date with the year, month, and day parts, working with time zones and calendars, and many others. Third-party libraries can implement these features and a recommended one is Howard Hinnant’s *date* library available under an MIT license at https://github.com/HowardHinnant/date.

See also

- *Measuring function execution time with a standard clock*

Measuring function execution time with a standard clock

In the previous recipe, we saw how to work with time intervals using the `chrono` standard library. However, we also often need to handle time points. The `chrono` library provides such a component, representing a duration of time since the epoch of a clock (that is, the beginning of time as defined by a clock). In this recipe, we will see how to use the `chrono` library and time points to measure the execution of a function.

Getting ready

This recipe is tightly related to the preceding one, *Expressing time intervals with chrono::duration*. If you did not go through that recipe before, you should do that before continuing with this one.

For the examples in this recipe, we will consider the following function that does nothing, but takes some time to execute:

```
void func(int const count = 100000000)
{
  for (int i = 0; i < count; ++i);
}
```

How to do it…

To measure the execution of a function, you must perform the following steps:

1. Retrieve the current moment of time using a standard clock:

```
auto start = std::chrono::high_resolution_clock::now();
```

2. Call the function you want to measure:

```
func();
```

3. Retrieve the current moment of time again; the difference between the two is the execution time of the function:

```
auto diff = std::chrono::high_resolution_clock::now() - start;
```

4. Convert the difference (that is expressed in nanoseconds) to the actual resolution you are interested in:

```
std::cout << std::chrono::duration<double,std::milli>(diff).count()
          << "ms" << std::endl;
std::cout << std::chrono::duration<double,std::nano>(diff).count()
          << "ns" << std::endl;
```

To implement this pattern in a reusable component, perform the following steps:

1. Create a class template parameterized with the resolution and the clock.
2. Create a static variadic function template that takes a function and its arguments.
3. Implement the pattern shown above, invoking the function with its arguments.
4. Return a duration, not the number of ticks.

```
template <typename Time = std::chrono::microseconds,
          typename Clock = std::chrono::high_resolution_clock>
struct perf_timer
{
  template <typename F, typename... Args>
  static Time duration(F&& f, Args... args)
  {
    auto start = Clock::now();

    std::invoke(std::forward<F>(f), std::forward<Args>(args)...);

    auto end = Clock::now();

    return std::chrono::duration_cast<Time>(end - start);
  }
};
```

How it works…

A clock is a component that defines two things:

- A beginning of time called *epoch*; there is no constraint of what the epoch is, but typical implementations use January 1, 1970.
- A *tick rate* that defines the increment between two time points (such as a millisecond or nanosecond).

A time point is a duration of time since the epoch of a clock. There are several time points that are of particular importance:

- The current time, returned by the clock's static member `now()`.
- The epoch, or the beginning of time; this is the time point created by the default constructor of `time_point` for a particular clock.
- The minimum time that can be represented by a clock, returned by the static member `min()` of `time_point`.
- The maximum time that can be represented with a clock, returned by the static member `max()` of a `time point`.

The standard defines three types of clocks:

- `system_clock`: This uses the real-time clock of the current system to represent time points.
- `high_resolution_clock`: This represents a clock that uses the shortest possible tick period on the current system.
- `steady_clock`: This indicates a clock that is never adjusted. This means that, unlike the other clocks, as the time advances, the difference between two time points is always positive.

The following example prints the precision of each clock, regardless of whether it is steady (or monotone) or not:

```
  template <typename T>
  void print_clock()
  {
    std::cout << "precision: "
              << (1000000.0 * double(T::period::num)) / (T::period::den)
              << std::endl;
    std::cout << "steady: " << T::is_steady << std::endl;
  }

  print_clock<std::chrono::system_clock>();
  print_clock<std::chrono::high_resolution_clock>();
  print_clock<std::chrono::steady_clock>();
```

A possible output is the following:

```
precision: 0.1
steady: 0
precision: 0.001
steady: 1
precision: 0.001
```

```
steady: 1
```

This means that the `system_clock` has a resolution of 0.1 milliseconds and is not a monotone clock. On the other hand, the other two clocks, `high_resolution_clock` and `steady_clock`, have both a resolution of 1 nanosecond and are monotone clocks.

The steadiness of a clock is important when measuring the execution time of a function, because if the clock is adjusted while the function runs, the result will not yield the actual execution time, and values can even be negative. You should rely on a steady clock to measure the function execution time. The typical choice for that is the `high_resolution_clock`, and that was the clock we used in the examples in the *How to do it…* section.

When we measure the execution time, we need to retrieve the current time before making the call and after the call returns. For that, we use the clock's `now()` static method. The result is a `time_point`; when we subtract two time points, the result is a `duration`, defined by the duration of the clock.

In order to create a reusable component that can be used to measure the execution time of any function, we have defined a class template called `perf_timer`. This class template is parameterized with the resolution we are interested in, which, by default, is microseconds, and the clock we want to use, which, by default, is `high_resolution_clock`. The class template has a single static member `duration()`—that is a variadic function template—that takes a function to execute and its variable number of arguments. The implementation is relatively simple: we retrieve the current time, invoke the function using `std::invoke` (so that it handles the different mechanisms for invoking anything callable), and then retrieve the current time again. The return value is a `duration` (with the defined resolution):

```
auto t = perf_timer<>::duration(func, 100000000);

std::cout << std::chrono::duration<double, std::milli>(t).count()
          << "ms" << std::endl;
std::cout << std::chrono::duration<double, std::nano>(t).count()
          << "ns" << std::endl;
```

It is important to note that we are not returning a number of ticks from the `duration()` function, but an actual `duration` value. The reason is that by returning a number of ticks we lose the resolution, and won't know what they actually represent. It is better to call `count()` only when the actual count of ticks is necessary:

```
auto t1 = perf_timer<std::chrono::nanoseconds>::duration(func1);
auto t2 = perf_timer<std::chrono::microseconds>::duration(func2);
auto t3 = perf_timer<std::chrono::milliseconds>::duration(func3);

std::cout
  << std::chrono::duration<double, std::micro>(t1 + t2 + t3).count()
  << "us" << std::endl;
```

See also

- *Expressing time intervals with chrono::duration*
- *Uniformly invoking anything callable* recipe of Chapter 3, Exploring functions

Generating hash values for custom types

The standard library provides several unordered associative containers: `std::unordered_set`, `std::unordered_multiset`, `std::unordered_map`, and `std::unordered_map`. These containers do not store their elements in a particular order; instead, they are grouped in buckets. The bucket an element belongs to depends on the hash value of the element. These standard containers use, by default, the `std::hash` class template to compute the hash value. The specialization for all basic types and also some library types is available. However, for custom types, you must specialize the class template yourself. This recipe will show you how to do that and also explain how a good hash value can be computed.

Getting ready

This recipe covers hashing functionalities from the standard library. You should be familiar with the concepts of hashes and hash functions.

For the examples in this recipe, we will use the following class:

```
struct Item
{
  int id;
  std::string name;
  double value;

  Item(int const id, std::string const & name, double const value)
    :id(id), name(name), value(value)
  {}

  bool operator==(Item const & other) const
  {
    return id == other.id && name == other.name &&
           value == other.value;
  }
};
```

How to do it…

In order to use your custom types with the unordered associative containers, you must perform the following steps:

1. Specialize the `std::hash` class template for your custom type; the specialization must be done in the `std` namespace.
2. Define synonyms for the argument and result type.
3. Implement the call operator so that it takes a constant reference to your type and returns a hash value.

To compute a good hash value, you should do the following:

1. Start with an initial value that should be a prime number (for example, 17).
2. For each field that is used to determine whether two instances of the class are equal, adjust the hash value according to the following formula:

```
hashValue = hashValue * prime + hashFunc(field);
```

3. You can use the same prime number for all fields with the above formula, but it is recommended to have a different value than the initial value (for instance, 31).
4. Use specialization of `std::hash` to determine the hash value for class data members.

Based on the steps described earlier, the `std::hash` specialization for class `Item` looks like this:

```
namespace std
{
  template<>
  struct hash<Item>
  {
    typedef Item argument_type;
    typedef size_t result_type;

    result_type operator()(argument_type const & item) const
    {
      result_type hashValue = 17;
      hashValue = 31 * hashValue +
                  std::hash<int>{}(item.id);
      hashValue = 31 * hashValue +
                  std::hash<std::string>{}(item.name);
      hashValue = 31 * hashValue +
                  std::hash<double>{}(item.value);

      return hashValue;
    }
  };
}
```

How it works…

The class template `std::hash` is a function object template whose call operator defines a hash function with the following properties:

- Takes an argument of the template parameter type and returns a `size_t` value.
- Does not throw any exceptions.
- For two arguments that are equal, it returns the same hash value.
- For two arguments that are not equal, the probability of returning the same value is very small (should be close to `1.0/std::numeric_limits<size_t>::max()`).

The standard provides specialization for all basic types, such as `bool`, `char`, `int`, `long`, `float`, `double` (with all the possible `unsigned` and `long` variations), and the pointer type, but also library types including the `basic_string` and `basic_string_view` types, `unique_ptr` and `shared_ptr`, `bitset` and `vector<bool>`, `optional` and `variant` (in C++17), and several other types. However, for custom types, you have to provide your own specialization. This specialization must be in the namespace `std` (because that is the namespace where the class template `hash` is defined) and must meet the requirements enumerated earlier.

The standard does not specify how hash values should be computed, and you can use any function you want as long as it returns the same value for equal objects and has a very small chance of returning the same value for non-equal objects. The algorithm described in this recipe was presented in the book *Effective Java 2nd Edition* by Joshua Bloch.

When computing the hash value, consider only the fields that participate in determining whether two instances of the class are equal (in other words, fields that are used in `operator==`). However, you must use all these fields that are used with `operator==`. In our example, all the three fields of class `Item` are used to determine the equality of two objects; therefore, we must use them all to compute the hash. The initial hash value should be nonzero, and in our example, we picked the prime number 17. The important thing is that these values should not be zero, otherwise initial fields (that is, first in the order of processing) that produce the hash value zero will not alter the hash (that remains zero as `x * 0 + 0 = 0`). For every field used in computing the hash, we alter the current hash by multiplying its previous value with a prime number and adding the hash of the current field. For this purpose, we use specializations of the class template `std::hash`. The use of prime 31 is advantageous for performance optimizations, because `31 * x` can be replaced by the compiler with `(x << 5) - x`, which is faster. Similarly, you can use 127, because `127 * x` is equal to `(x << 7) - x` or 8191, because `8191 * x` is equal to `(x << 13) - x`.

If your custom type contains an array and is used to determine the equality of two objects and, therefore, needs to be used to compute the hash, then treat the array as if its elements were data members of the class. In other words, apply the same algorithm described earlier for all elements of the array.

Having the specialization `std::hash<Item>` shown in the *How to do it…* section, we can use the `Item` class with unordered associative containers, such as `std::unordered_set`:

```
  std::unordered_set<Item> set2
  {
    { 1, "one"s, 1.0 },
    { 2, "two"s, 2.0 },
    { 3, "three"s, 3.0 },
  };
```

Using std::any to store any value

C++ does not have a hierarchical type system like other languages (such as C# or Java) and, therefore, it does not have a possibility to store multiple types of value in a single variable like it is possible with type `Object` in .NET and Java or natively in JavaScript. Developers have long time used `void*` for that purpose, but this only helps store pointers to anything and is not type-safe. Depending on the end goal, alternatives can include templates or overloaded functions. However, C++17 has introduced a standard type-safe container, called `std::any`, that can hold a single value of any type.

Getting ready

`std::any` has been designed based on `boost::any` and is available in the `<any>` header. If you are familiar with `boost::any` and have used it in your code, you can migrate it seamlessly to `std::any`.

How to do it…

Use the following operations to work with `std::any`:

- To store values, use the constructor or assign them directly to a `std::any` variable:

```
std::any value(42); // integer 12
value = 42.0;       // double 12.0
value = "42"s;      // std::string "12"
```

- To read values, use the non-member function `std::any_cast()`:

```
std::any value = 42.0;
try
{
  auto d = std::any_cast<double>(value);
  std::cout << d << std::endl;
}
catch (std::bad_any_cast const & e)
{
  std::cout << e.what() << std::endl;
}
```

- To check the type of the stored value, use the member function `type()`:

```
inline bool is_integer(std::any const & a)
{
  return a.type() == typeid(int);
}
```

- To check whether the container stores a value, use the `has_value()` member function:

```
auto ltest = [](std::any const & a) {
  if (a.has_value())
    std::cout << "has value" << std::endl;
  else
    std::cout << "no value" << std::endl;
};

std::any value;
ltest(value); // no value
value = 42;
ltest(value); // has value
```

- To modify the stored value, use member functions `emplace()`, `reset()`, or `swap()`:

```
std::any value = 42;
ltest(value); // has value
value.reset();
ltest(value); // no value
```

How it works…

`std::any` is a type-safe container that can hold values of any type that is (or rather whose decayed type is) copy constructible. Storing values in the container is very simple—you can either use one of the available constructors (the default constructor creates a container that stores no value) or the assignment operator. However, reading values is not directly possible, and you need to use the non-member function `std::any_cast()` that casts the stored value to the specified type. This function throws `std::bad_any_cast` if the stored value has a different type than the one you are casting to. Casting between implicitly convertible types, such as `int` and `long`, is not possible either. `std::bad_any_cast` is derived from `std::bad_cast`; therefore, you can catch any of these two exception types.

It is possible to check the type of the stored value using the `type()` member function that returns a `type_info` constant reference. If the container is empty, this function returns `typeid(void)`. To check whether the container stores a value, you can use the member function `has_value()` that returns `true` if there is a value or `false` if the container is empty.

The following example shows how to check whether the container has any value, how to check the type of the stored value, and how to read the value from the container:

```
void log(std::any const & value)
{
  if (value.has_value())
  {
    auto const & tv = value.type();
    if (tv == typeid(int))
    {
      std::cout << std::any_cast<int>(value) << std::endl;
    }
    else if (tv == typeid(std::string))
    {
      std::cout << std::any_cast<std::string>(value) << std::endl;
    }
    else if (tv == typeid(
       std::chrono::time_point<std::chrono::system_clock>))
    {
      auto t = std::any_cast<std::chrono::time_point<
                  std::chrono::system_clock>>(value);
      auto now = std::chrono::system_clock::to_time_t(t);
      std::cout << std::put_time(std::localtime(&now), "%F %T")
                << std::endl;
    }
    else
    {
      std::cout << "unexpected value type" << std::endl;
    }
  }
  else
  {
    std::cout << "(empty)" << std::endl;
  }
}

log(std::any{});                      // (empty)
log(12);                              // 12
log("12"s);                           // 12
log(12.0);                            // unexpected value type
log(std::chrono::system_clock::now()); // 2016-10-30 22:42:57
```

If you want to store multiple values of any type, use a standard container such as `std::vector` to hold values of the type `std::any`:

```
  std::vector<std::any> values;
  values.push_back(std::any{});
  values.push_back(12);
  values.push_back("12"s);
  values.push_back(12.0);
  values.push_back(std::chrono::system_clock::now());

  for (auto const v : values)
    log(v);
```

See also

- *Using std::optional to store optional values*
- *Using std::variant as a type-safe union*

Using std::optional to store optional values

Sometimes, it is useful to be able to store either a value or a null if a value is not available. A typical example for such a case is the return value of a function that may fail to produce a return value, but this failure is not an error. For instance, think of a function that finds and returns values from a dictionary by specifying a key. Not finding a value is a probable case and, therefore, the function would either return a Boolean (or an integer value, if more error codes are necessary) and have a reference argument to hold the return value or return a pointer (raw or smart pointer). In C++17, `std::optional` is a better alternative to these solutions. The class template `std::optional` is a template container for storing a value that may or may not exist. In this recipe, we will see how to use this container and what are its typical use cases.

Getting ready

The class template `std::optional<T>` was designed based on `boost::optional` and is available in the `<optional>` header. If you are familiar with `boost::optional` and have used it in your code, you can migrate it seamlessly to `std::optional`.

How to do it…

Use the following operations to work with `std::optional`:

- To store a value, use the constructor or assign the value directly to an `std::optional` object:

```
std::optional<int> v1;      // v1 is empty
std::optional<int> v2(42);  // v2 contains 42
v1 = 42;                    // v1 contains 42
std::optional<int> v3 = v2; // v3 contains 42
```

- To read the stored value, use `operator*` or `operator->`:

```
std::optional<int> v1{ 42 };
std::cout << *v1 << std::endl;   // 42
std::optional<foo> v2{ foo{ 42, 10.5 } };
std::cout << v2->a << ", "
          << v2->b << std::endl; // 42, 10.5
```

- Alternatively, use member functions `value()` and `value_or()` to read the stored value:

```
std::optional<std::string> v1{ "text"s };
std::cout << v1.value()
          << std::endl; // text

std::optional<std::string> v2;
std::cout << v2.value_or("default"s)
          << std::endl; // default
```

- To check whether the container stores a value, use a conversion operator to `bool` or the member function `has_value()`:

```
struct foo
{
  int a;
  double b;
};

std::optional<int> v1{ 42 };
if (v1) std::cout << *v1 << std::endl;

std::optional<foo> v2{ foo{ 42, 10.5 } };
if (v2.has_value())
  std::cout << v2->a << ", " << v2->b << std::endl;
```

- To modify the stored value, use member functions `emplace()`, `reset()`, or `swap()`:

```
std::optional<int> v{ 42 }; // v contains 42
v.reset();                  // v is empty
```

Use `std::optional` to model any of the following:

- Return values from functions that may fail to produce a value:

```
template <typename K, typename V>
std::optional<V> find(int const key,
                      std::map<K, V> const & m)
{
  auto pos = m.find(key);
  if (pos != m.end())
```

```
    return pos->second;
  return {};
}

std::map<int, std::string> m{
   { 1, "one"s },{ 2, "two"s },{ 3, "three"s } };

auto value = find(2, m);
if (value) std::cout << *value << std::endl; // two

value = find(4, m);
if (value) std::cout << *value << std::endl;
```

- Parameters to functions that are optional:

```
std::string extract(std::string const & text,
                    std::optional<int> start,
                    std::optional<int> end)
{
  auto s = start.value_or(0);
  auto e = end.value_or(text.length());
  return text.substr(s, e - s);
}

auto v1 = extract("sample"s, {}, {});
std::cout << v1 << std::endl; // sample

auto v2 = extract("sample"s, 1, {});
std::cout << v2 << std::endl; // ample

auto v3 = extract("sample"s, 1, 4);
std::cout << v3 << std::endl; // amp
```

- Class data members that are optional:

```
struct book
{
  std::string                title;
  std::optional<std::string> subtitle;
  std::vector<std::string>   authors;
  std::string                publisher;
  std::string                isbn;
  std::optional<int>         pages;
  std::optional<int>         year;
};
```

How it works…

The class template `std::optional` is a class template that represents a container for an optional value. If the container does have a value, that value is stored as part of the `optional` object; no heap allocations and pointers are involved. The `std::optional` class template is conceptually implemented like this:

```
template <typename T>
class optional
{
  bool _initialized;
  std::aligned_storage_t<sizeof(t), alignof(T)> _storage;
};
```

The `std::aligned_storage_t` alias template allows us to create uninitialized chunks of memory that can hold objects of a given type. The class template `std::optional` does not contain a value if it was default constructed, or if it was copy constructed or copy assigned from another empty optional object or from an `std::nullopt_t` value. This is a helper type, implemented as an empty class, that indicates an optional object with an uninitialized state.

The typical use for an `optional` type (called *nullable* in other programming languages) is the return type from a function that may fail. Possible solutions for this situation include the following:

- Return an `std::pair<T, bool>`, where `T` is the type of the return value; the second element of the pair is a Boolean flag that indicates whether the value of the first element is valid or not.
- Return a `bool` and take an extra parameter of type `T&` and assign a value to this parameter only if the function succeeds.
- Return a raw or smart pointer type, and use `nullptr` to indicate a failure.

The class template `std::optional` is a better approach because, on one hand, it does not involve output parameters to the function (which is unnatural for returning values) and does not require working with pointers, and, on the other hand, it better encapsulates the details of an `std::pair<T, bool>`. However, optional objects can also be used for class data members, and compilers are able to optimize the memory layout for an efficient storage.

The class template `std::optional` cannot be used to return polymorphic types. If you write, for instance, a factory method that needs to return different types from a hierarchy of types, you cannot rely on `std::optional` and need to return a pointer, preferably a `std::shared_ptr` or `std::unique_ptr` (depending if ownership of the object needs to be shared or not).

When you use `std::optional` to pass optional arguments to a function, you need to understand that it may incur creating copies, which can be a performance issue if large objects are involved. Let's consider the following example of a function that has a constant reference to the `std::optional` parameter:

```
struct bar { /* details */ };
```

```
void process(std::optional<bar> const & arg)
{
  /* do something with arg */
}

std::optional<bar> b1{ bar{} };
bar b2{};

process(b1); // no copy
process(b2); // copy construction
```

The first call to `process()` does not involve any additional object construction because we pass an `std::optional<bar>` object. The second call, however, will involve the copy construction of a `bar` object, because `b2` is a `bar` and needs to be copied to an `std::optional<bar>`; a copy is made even if `bar` has move semantics implemented. If `bar` was a small object, this shouldn't be of a great concern, but for large objects, it can prove a performance issue. The solution to avoid this depends on the context, and can involve creating a second overload that takes a constant reference to `bar`, or entirely avoiding using `std::optional`.

See also

- *Using std::any to store any value*
- *Using std::variant as a type-safe union*

Using std::variant as a type-safe union

In C++, union is a special class type that, at any point, holds a value of one of its data members. Unlike regular classes, unions cannot have base classes nor can they be derived, and they cannot contain virtual functions (that would not make sense anyway). Unions are mostly used to define different representations of the same data. However, unions only work for types that are POD. If a union contains values of non-POD types, then these members require explicit construction with a placement `new` and explicit destruction, which is cumbersome and error-prone. In C++17, a type-safe union is available in the form of a standard library class template called `std::variant`. In this recipe, you will learn how to use it to model alternative values.

Getting ready

Although discriminated unions are not directly discussed in this recipe, being familiar with them will help understand better the design of, and the way `variant` works.

The class template `std::variant` was designed based on `boost::variant` and is available in the `<variant>` header. If you are familiar with `boost::variant` and have used it in your code, you can migrate your code with little effort to use the standard `variant` class template.

How to do it…

Use the following operations to work with `std::variant`:

- To modify the stored value, use member functions `emplace()` or `swap()`:

```
struct foo
{
  int value;
  explicit foo(int const i) : value(i) {}
};

std::variant<int, std::string, foo> v = 42; // holds int
v.emplace<foo>(42);                         // holds foo
```

- To read the stored values, use non-member functions `std::get` or `std::get_if`:

```
std::variant<int, double, std::string> v = 42;

auto i1 = std::get<int>(v);
auto i2 = std::get<0>(v);

try
{
  auto f = std::get<double>(v);
}
catch (std::bad_variant_access const & e)
{
  std::cout << e.what() << std::endl; // Unexpected index
}
```

- To store a value, use the constructor or assign a value directly to a variant object:

```
std::variant<int, double, std::string> v;
v = 42;   // v contains int 42
v = 42.0; // v contains double 42.0
v = "42"; // v contains string "42"
```

- To check what is the stored alternative, use member function `index()`:

```
std::variant<int, double, std::string> v = 42;
static_assert(std::variant_size_v<decltype(v)> == 3);
std::cout << "index = " << v.index() << std::endl;
v = 42.0;
std::cout << "index = " << v.index() << std::endl;
v = "42";
std::cout << "index = " << v.index() << std::endl;
```

- To check whether a variant holds an alternative, use the non-member function `std::holds_alternative()`:

```
std::variant<int, double, std::string> v = 42;
std::cout << "int? " << std::boolalpha
          << std::holds_alternative<int>(v)
          << std::endl; // int? true

v = "42";
std::cout << "int? " << std::boolalpha
          << std::holds_alternative<int>(v)
          << std::endl; // int? false
```

- To define a variant whose first alternative is not default constructible, use

`std::monostate` as the first alternative (in this example, `foo` is the same class as earlier):

```
std::variant<std::monostate, foo, int> v;
v = 42;        // v contains int 42
std::cout << std::get<int>(v) << std::endl;
v = foo{ 42 }; // v contains foo{42}
std::cout << std::get<foo>(v).value << std::endl;
```

- To process the stored value of a variant and do something depending on the type of the alternative, use `std::visit()`:

```
std::variant<int, double, std::string> v = 42;
std::visit(
  [](auto&& arg) {std::cout << arg << std::endl; },
  v);
```

How it works…

`std::variant` is a class template that models a type-safe union, holding a value of one of its possible alternatives at any given time. In some rare cases, it is possible, though, that a variant object does not store any value. `std::variant` has a member function called `valueless_by_exception()` that returns `true` if the variant does not hold a value, which is possible only in case of an exception during initialization, therefore, the name of the function.

The size of an `std::variant` object is as large as its largest alternative. A variant does not store additional data. The value stored by the variant is allocated within the memory representation of the object itself.

A variant can hold multiple alternatives of the same type, and also to hold different constant- and volatile-qualified versions of the same time. On the other hand, it cannot hold an alternative of type `void`, or alternatives of array and reference types. On the other hand, the first alternative must always be default constructible. The reason for that is that, just like discriminated unions, a variant is default initialized with the value of its first alternative. If the first alternative type is not default constructible, then the variant must use `std::monostate` as the first alternative. This is an empty type indented for making variants default constructible.

It is possible to query a `variant` at compile time for its size (that is, the number of alternatives it defines) and for the type of an alternative specified by its zero-based index. On the other hand, you can query the index of the currently hold alternative at runtime using the member function `index()`.

There’s more…

A typical way of manipulating the content of a variant is through visitation. This is basically the execution of an action based on the alternative hold by the variant. Since it is a larger topic, it is addressed separately in the next recipe.

See also

- *Using std::any to store any value*
- *Using std::optional to store optional values*
- *Visiting a std::variant*

Visiting a std::variant

`std::variant` is a new standard container added to C++17 based on the `boost.variant` library. A variant is a type-safe union that holds the value of one of its alternative types. Although in the previous recipe we have seen various operations with variants, the variants we used were rather simple, with POD types mostly, which is not the actual purpose for which `std::variant` was created. Variants are intended to be used for holding alternatives of similar non-polymorphic and non-POD types. In this recipe, we will see a more real-world example of using variants and will learn how to visit variants.

Getting ready

For this recipe, you should be familiar with the `std::variant` type. It is recommended that you first read the previous recipe, *Using std::variant as a type-safe union*.

To explain how variant visitation can be done, we will consider a variant for representing a media DVD. Let's suppose we want to model a store or library that has DVDs that could contain either music, a movie, or software. However, these options are not modeled as a hierarchy with common data and virtual functions, but rather as non-related types that may have similar properties, such as a title. For simplicity, we consider the following properties:

- For a movie: Title and length (in minutes)
- For an album: Title, artist name, and a list of tracks (each track having a title and length in seconds)
- For software: Title and manufacturer

The following shows a simple implementation of these types, without any functions, because that is not relevant to the visitation of a variant holding alternatives of these types:

```
enum class Genre { Drama, Action, SF, Commedy };

struct Movie
{
  std::string title;
  std::chrono::minutes length;
  std::vector<Genre> genre;
};

struct Track
{
  std::string title;
  std::chrono::seconds length;
};

struct Music
{
  std::string title;
  std::string artist;
  std::vector<Track> tracks;
};

struct Software
{
  std::string title;
  std::string vendor;
};

using dvd = std::variant<Movie, Music, Software>;
```

How to do it…

To visit a variant, you must provide one or more actions for the possible alternatives of the variant. There are several types of visitors that are used for different purposes:

- A void visitor that does not return anything, but has side-effects. The following example prints the title of each DVD to the console:

```
for (auto const & d : dvds)
{
  std::visit([](auto&& arg) {
               std::cout << arg.title << std::endl; },
             d);
}
```

- A visitor that returns a value; the value should have the same type regardless of the current alternative of the variant, or can be itself a variant. In the following example, we visit a variant and return a new variant of the same type that has the title property from any of its alternatives transformed to uppercase letters:

```
for (auto const & d : dvds)
{
  dvd result = std::visit(
      [](auto&& arg) -> dvd
      {
        auto cpy { arg };
        cpy.title = to_upper(cpy.title);
        return cpy;
      },
    d);

  std::visit(
    [](auto&& arg) {
       std::cout << arg.title << std::endl; },
    result);
}
```

- A visitor that does type matching (which can either be a void or a value-returning visitor) implemented by providing a function object that has an overloaded call operator for each alternative type of the variant:

```
struct visitor_functor
{
  void operator()(Movie const & arg) const
  {
    std::cout << "Movie" << std::endl;
    std::cout << " Title: " << arg.title << std::endl;
    std::cout << " Length: " << arg.length.count()
              << "min" << std::endl;
  }

  void operator()(Music const & arg) const
  {
    std::cout << "Music" << std::endl;
    std::cout << " Title: " << arg.title << std::endl;
    std::cout << " Artist: " << arg.artist << std::endl;

    for (auto const & t : arg.tracks)
      std::cout << " Track: " << t.title
                << ", " << t.length.count()
                << "sec" << std::endl;
  }
```

```
  void operator()(Software const & arg) const
  {
    std::cout << "Software" << std::endl;
    std::cout << " Title: " << arg.title << std::endl;
    std::cout << " Vendor: " << arg.vendor << std::endl;
  }
};

for (auto const & d : dvds)
{
  std::visit(visitor_functor(), d);
}
```

- A visitor that does type matching implemented by providing a lambda expression that performs an action based on the type of the alternative:

```
for (auto const & d : dvds)
{
  std::visit([](auto&& arg) {
    using T = std::decay_t<decltype(arg)>;
    if constexpr (std::is_same_v<T, Movie>)
    {
      std::cout << "Movie" << std::endl;
      std::cout << " Title: " << arg.title << std::endl;
      std::cout << " Length: " << arg.length.count()
                << "min" << std::endl;
    }
    else if constexpr (std::is_same_v<T, Music>)
    {
      std::cout << "Music" << std::endl;
      std::cout << " Title: " << arg.title << std::endl;
      std::cout << " Artist: " << arg.artist << std::endl;

      for (auto const & t : arg.tracks)
        std::cout << " Track: " << t.title
                  << ", " << t.length.count()
                  << "sec" << std::endl;
    }
    else if constexpr (std::is_same_v<T, Software>)
    {
      std::cout << "Software" << std::endl;
      std::cout << " Title: " << arg.title << std::endl;
      std::cout << " Vendor: " << arg.vendor << std::endl;
    }
  },
  d);
}
```

How it works…

A visitor is a callable object (a function, a lambda expression, or a function object) that accepts every possible alternative from a variant. Visitation is done by invoking `std::visit()` with the visitor and one or more variant objects. The variants do not have to be of the same type, but the visitor must be able to accept every possible alternative from all the variants it is invoked for. In the examples earlier, we have visited a single variant object, but visiting multiple variants does not imply anything more than passing them as arguments to `std::visit()`.

When you visit a variant, the callable object is invoked with the value currently stored in the variant. If the visitor does not accept an argument of the type stored in the variant, the program is ill-formed. If the visitor is a function object, then it must overload its call operator for all the possible alternative types of the variant. If the visitor is a lambda expression, it should be a generic lambda, which is basically a function object with a call operator template, instantiated by the compiler with the actual type that it is invoked with.

Examples of both approaches were shown in the previous section for a type-matching visitor. The function object in the first example is straightforward and should not require additional explanations. On the other hand, the generic lambda expression uses `constexpr if` to select a particular `if` branch based on the type of the argument at compile time. The result is that the compiler will create a function object with an operator call template and a body that contains `constexpr if` statements; when it instantiates that function template, it will produce an overload for each possible alternative type of the variant, and in each of these overloads, it will select only the `constexpr if` branch that matches the type of the call operator argument. The result is conceptually equivalent to the implementation of the `visitor_functor` class.

See also

- *Using std::any to store any value*
- *Using std::optional to store optional values*
- *Using std::variant as a type-safe union*

Registering a function to be called when a program exits normally

It is common that a program, upon exit, must perform cleanup code to release resources, or write something to a log, or do some other end operation. The standard library provides two utility functions that enable us to register functions to be called when a program terminates normally, either by returning from `main()` or through a call to `std::exit()` or `std::quick_exit()`. This is particularly useful for libraries that need to perform an action before the program is terminated, without relying on the user to explicitly call an end function. In this recipe, you will learn how to install exit handlers and how they work.

Getting ready

All the functions discussed in this recipe, `exit()`, `quick_exit()`, `atexit()`, and `at_quick_exit()`, are available in the namespace `std` in the header `<cstdlib>`.

How to do it…

To register functions to be called upon termination of a program, you should use the following:

- `std::atexit()` to register functions to be invoked when they return from `main()` or when a call to `std::exit()` is made:

```
void exit_handler_1()
{
  std::cout << "exit handler 1" << std::endl;
}

void exit_handler_2()
{
  std::cout << "exit handler 1" << std::endl;
}

std::atexit(exit_handler_1);
std::atexit(exit_handler_2);
std::atexit([]() {std::cout << "exit handler 3" << std::endl; });
```

- `std::at_quick_exit()` to register functions to be invoked when a call to `std::quick_exit()` is made:

```
void quick_exit_handler_1()
{
  std::cout << "quick exit handler 1" << std::endl;
}

void quick_exit_handler_2()
{
  std::cout << "quick exit handler 2" << std::endl;
}

std::at_quick_exit(quick_exit_handler_1);
std::at_quick_exit(quick_exit_handler_2);
std::at_quick_exit([]() {
  std::cout << "quick exit handler 3" << std::endl; });
```

How it works…

The exit handlers, regardless of the method they are registered with, are called only when the program terminates normally or quickly. If termination is done in an abnormal way, via a call to `std::terminate()` or `std::abort()`, none of them are called. If any of these handlers exits via an exception, then `std::terminate()` is called. Exit handlers must not have any parameters and must return `void`. Once registered, an exit handler cannot be unregistered.

A program can install multiple handlers. The standard guarantees that at least 32 handlers can be registered with each method, although actual implementations can support any higher number. Both `std::atexit()` and `std::at_quick_exit()` are thread-safe and, therefore, can be called simultaneously from different threads without incurring race conditions.

If multiple handlers are registered, then they are called in the reverse order of the registration. The following table shows the output of a program that registered the exit handlers, as shown in the previous section, when the program terminates via an `std::exit()` and an `std::quick_exit()` call:

`std::exit(0);`	`std::quick_exit(0);`
exit handler 3 exit handler 2 exit handler 1	quick exit handler 3 quick exit handler 2 quick exit handler 1

On the other hand, on normal termination of the program, destruction of objects with local storage duration, destruction of objects with static storage duration, and call of registered exit handlers are done concurrently. However, it is guaranteed that exit handlers registered before the construction of a static object are called after the destruction of that static object, and exit handlers registered after the construction of a static object are called before the destruction of that static object. To better exemplify this, let's consider the following class:

```
struct static_foo
{
  ~static_foo() { std::cout << "static foo destroyed!" << std::endl; }
  static static_foo* instance()
  {
    static static_foo obj;
    return &obj;
  }
};
```

When the following sequence of code is executed, `exit_handler_1` is registered before the creation of the static object `static_foo`. On the other hand, `exit_handler_2` and the lambda expression are both registered, in that order, after the static object was constructed. As a result, the order of calls at normal termination is as follows:

1. Lambda expression
2. `exit_handler_2`
3. Destructor of `static_foo`
4. `exit_handler_1`

```
    std::atexit(exit_handler_1);
    static_foo::instance();
    std::atexit(exit_handler_2);
    std::atexit([]() {std::cout << "exit handler 3" << std::endl; });

    std::exit(42);
```

The output for the preceding program is as follows:

```
exit handler 3
exit handler 2
static foo destroyed!
exit handler 1
```

See also

- *Using lambdas with standard algorithms* recipe of Chapter 3, Exploring functions

Using type traits to query properties of types

Template metaprogramming is a powerful feature of the language that enables us to write and reuse generic code that works with all types. In practice, however, it is often necessary that generic code should work differently, or not at all, with different types, either through intent, or for semantic correctness, performance, or other reasons. For example, you may want a generic algorithm to be implemented differently for POD and non-POD types, or you want a function template to be instantiated only with integral types. C++11 provides a set of type traits to help with this. Type traits are basically meta-types that provide information about other types. The type traits library contains a long list of traits for querying type properties (such as checking whether a type is an integral type or whether two types are the same), but also for performing type transformation (such as removing `const` and `volatile` qualifiers or adding a pointer to a type). We have used type traits in several recipes earlier in the book; however, in this recipe, we will look into what the type traits are and how they work.

Getting ready

All type traits, introduced in C++11, are available in the namespace `std` in the `<type_traits>` header.

Type traits can be used in many metaprogramming contexts, and throughout the book, we have seen them used in various situations. In this recipe, we will summarize some of these use cases and see how type traits work.

In this recipe, we will discuss full and partial template specialization. Familiarity with these concepts will help you better understand the way type traits work.

How to do it…

The following list shows various situations where type traits are used to achieve various design goals:

- With `enable_if`, to define preconditions for the types a function template can be instantiated with:

```
template <typename T,
          typename = typename std::enable_if<
                std::is_arithmetic<T>::value>::type>
T multiply(T const t1, T const t2)
{
  return t1 * t2;
}

auto v1 = multiply(42.0, 1.5);     // OK
auto v2 = multiply("42"s, "1.5"s); // error
```

- With `static_assert`, to ensure that invariants are met:

```
template <typename T>
struct pod_wrapper
{
  static_assert(std::is_pod<T>::value, "Type is not a POD!");
  T value;
};

pod_wrapper<int> i{ 42 };            // OK
pod_wrapper<std::string> s{ "42"s }; // error
```

- With `std::conditional`, to select between types:

```
template <typename T>
struct const_wrapper
{
  typedef typename std::conditional<
    std::is_const<T>::value,
    T,
    typename std::add_const<T>::type>::type const_type;
};

static_assert(
  std::is_const<const_wrapper<int>::const_type>::value);

static_assert(
  std::is_const<const_wrapper<int const>::const_type>::value);
```

- With `constexpr if`, to enable the compiler to generate different code based on the type the template is instantiated with:

```
template <typename T>
auto process(T arg)
{
  if constexpr (std::is_same<T, bool>::value)
    return !arg;
  else if constexpr (std::is_integral<T>::value)
    return -arg;
  else if constexpr (std::is_floating_point<T>::value)
    return std::abs(arg);
  else
    return arg;
}
```

```
        auto v1 = process(false); // v1 = true
        auto v2 = process(42);    // v2 = -42
        auto v3 = process(-42.0); // v3 = 42.0
        auto v4 = process("42"s); // v4 = "42"
```

How it works…

Type traits are classes that provide meta-information about types or can be used to modify types. There are actually two categories of type traits:

- Traits that provide information about types, their properties, or their relations (such as `is_integer`, `is_arithmetic`, `is_array`, `is_enum`, `is_class`, `is_const`, `is_pod`, `is_constructible`, `is_same`, and so on). These traits provide a constant `bool` member called `value`.
- Traits that modify properties of types (such as `add_const`, `remove_const`, `add_pointer`, `remove_pointer`, `make_signed`, `make_unsigned`, and so on). These traits provide a member `typedef` called `type` that represents the transformed type.

Both of these categories of types have been shown in the *How to do it…* section; examples have been discussed and explained in detail in other recipes. For convenience, a short summary is provided here:

- In the first example, the function template `multiply()` is allowed to be instantiated only with arithmetic types (that is, integral or floating point); when instantiated with a different kind of type, `enable_if` does not define a `typedef` member called `type` and that produces a compilation error.
- In the second example, `pod_wrapper` is a class template that is supposed to be instantiated only with POD types. A `static_assert` declaration produces a compilation error if a non-POD type is used.
- In the third example, `const_wrapper` is a class template that provides a `typedef` member called `const_type` that represents a const-qualified type. In this example, we used `std::conditional` to select between two types at compile time: if the type parameter `T` is already a const type, then we just select `T`. Otherwise, we use the `add_const` type trait to qualify the type with the `const` specifier.
- If the fourth example, `process()` is a function template that contains a series of `if constexpr` branches. Based on the category of type, queried at compile time with various type traits (`is_same`, `is_integer`, `is_floating_point`), the compiler selects one branch only to be put into the generated code and discards the rest. Therefore, a call such as `process(42)` will produce the following instantiation of the function template:

```
int process(int arg)
{
  return -arg;
}
```

Type traits are implemented by providing a class template and a partial or full specialization for it. The following represent conceptual implementation for some type traits:

- The `is_void()` method indicates whether a type is `void`; this uses full specialization:

```
template <typename T>
struct is_void
```

```
    { static const bool value = false; };

    template <>
    struct is_void<void>
    { static const bool value = true; };
```

- The `is_pointer()` method indicates whether a type is a pointer to an object or a pointer to a function; this uses partial specialization:

```
    template <typename T>
    struct is_pointer
    { static const bool value = false; };

    template <typename T>
    struct is_pointer<T*>
    { static const bool value = true; };
```

There’s more…

Type traits are not limited to what the standard library provides. Using similar techniques, you can define your own type traits to achieve various goals. In the next recipe, we will see how we can define and use our own type traits.

See also

- *Selecting branches at compile time with constexpr if* recipe of Chapter 4, *Preprocessor and Compilation*
- *Conditionally compiling classes and functions with enable_if* recipe of Chapter 4, *Preprocessor and Compilation*
- *Performing compile-time assertion checks with static_assert* recipe of Chapter 4, *Preprocessor and Compilation*
- *Writing your own type traits*
- *Using std::conditional to choose between types*

Writing your own type traits

In the previous recipe, we have seen what type traits are, what traits the standard provides, and how they can be used for various purposes. In this recipe, we take a step further and take a look at how to define our own custom traits.

Getting ready

It is recommended that you first read the recipe, *Using type traits to query properties of types*, before you continue with this one.

In this recipe, we will learn how to solve the following problem: we have several classes that support serialization. Without getting into any details, let's suppose some provide a "plain" serialization to a string (regardless of what that can mean), whereas others do it based on a specified encoding. The end goal is to create a single, uniform API for serializing objects of any of these types. For this, we will consider the following two classes: `foo` that provides a simple serialization, and `bar` that provides serialization with encoding:

```
struct foo
{
  std::string serialize()
  {
    return "plain"s;
  }
};

struct bar
{
  std::string serialize_with_encoding()
  {
    return "encoded"s;
  }
};
```

How to do it…

Implement the following class and function templates:

- A class template called `is_serializable_with_encoding` containing a `static const bool` variable set to `false`:

```
template <typename T>
struct is_serializable_with_encoding
{
  static const bool value = false;
};
```

- A full specialization of the `is_serializable_with_encoding` template for class `bar` that has the `static const bool` variable set to `true`:

```
template <>
struct is_serializable_with_encoding<bar>
{
  static const bool value = true;
};
```

- A class template called `serializer`, containing a static template method called `serialize`, that takes an argument of the template type `T` and calls `serialize()` for that object:

```
template <bool b>
struct serializer
{
  template <typename T>
  static auto serialize(T& v)
  {
    return v.serialize();
  }
};
```

- A full specialization class template for `true`, whose `serialize()` static method calls `serialize_with_encoding()` for the argument:

```
template <>
struct serializer<true>
{
  template <typename T>
  static auto serialize(T& v)
  {
    return v.serialize_with_encoding();
  }
};
```

- A function template called `serialize()`, that uses the `serializer` class templates defined above and the `is_serializable_with_encoding` type trait to select which of the actual serialization methods (plain or with encoding) should be called:

```
template <typename T>
auto serialize(T& v)
{
  return serializer<is_serializable_with_encoding<T>::value>::
     serialize(v);
}
```

How it works…

`is_serializable_with_encoding` is a type trait that checks whether a type `T` is serializable with (a specified) encoding. It provides a static member of type `bool` called `value` that is equal to `true` if `T` supports serialization with encoding, or `false` otherwise. It is implemented as a class template with a single type template parameter `T`; this class template is fully specialized for the types that support encoded serialization, in this particular example, for the class `bar`:

```
std::cout <<
  is_serializable_with_encoding<foo>::value << std::endl;    // false
std::cout <<
  is_serializable_with_encoding<bar>::value << std::endl;    // true
std::cout <<
  is_serializable_with_encoding<int>::value << std::endl;    // false
std::cout <<
  is_serializable_with_encoding<string>::value << std::endl; // false
```

The `serialize()` method is a function template that represents a common API for serializing objects that support either type of serialization. It takes a single argument of the type template parameter `T` and uses a helper class template `serializer` to call either the `serialize()` or the `serialize_with_encoding()` method of its argument.

The `serializer` is a class template with a single, non-type template parameter of the type `bool`. This class template contains a static function template called `serialize()`. This function template takes a single parameter of the type template parameter `T`, calls `serialize()` on the argument, and returns the value returned from that call. The `serializer` class template has a full specialization for the value `true` of its non-type template parameter. In this specialization, the function template `serialize()` has an unchanged signature, but calls `serialize_with_encoding()` instead of `serialize()`.

The selection between using the generic or the fully specialized class template is done in the `serialize()` function template using the `is_serializable_with_encoding` type trait. The static member `value` of the type trait is used as the argument for the non-type template parameter of `serializer`.

With all that defined, we can write the following code:

```
foo f;
bar b;

std::cout << serialize(f) << std::endl; // plain
std::cout << serialize(b) << std::endl; // encoded
```

See also

- *Using type traits to query properties of types*
- *Using std::conditional to choose between types*

Using std::conditional to choose between types

In the previous recipes, we have looked at some of the features from the type support library, and particularly type traits. Related topics have been discussed in other parts of the book, such as using `std::enable_if` to hide function overloads, in Chapter 4, *Preprocessor and Compilation*, and `std::decay` to remove `const` and `volatile` qualifiers, when we discussed visiting variants, also in this chapter. Another type transformation feature worth discussing to a larger extent is `std::conditional` that enables us to choose between two types at compile time, based on a compile-time Boolean expression. From this recipe, you will learn how it works and how to use it through several examples.

Getting ready

It is recommended that you first read the *Using type traits to query properties of types* recipe of this chapter.

How to do it…

The following is a list of examples that show how to use `std::conditional` (and `std::conditional_t`) to choose at compile time between two types:

- In a type alias or `typedef`, to select between a 32-bit and 64-bit integer type, based on the platform (pointer size is 4 bytes on 32-bit platforms and 8 bytes on 68-bit platforms):

```
using long_type =
  std::conditional<
    sizeof(void*) <= 4,
    long,
    long long>::type;

auto n = long_type{ 42 };
```

- In an alias template, to select between a 8-, 16-, 32-, or 64-bit integer type, based on the user specification (as a non-type template parameter):

```
template <int size>
using number_type =
  typename std::conditional<
  size<=1,
  std::int8_t,
    typename std::conditional<
    size<=2,
    std::int16_t,
      typename std::conditional<
      size<=4,
      std::int32_t,
      std::int64_t
      >::type
    >::type
  >::type;

auto n = number_type<2>{ 42 };

static_assert(sizeof(number_type<1>) == 1);
static_assert(sizeof(number_type<2>) == 2);
static_assert(sizeof(number_type<3>) == 4);
static_assert(sizeof(number_type<4>) == 4);
static_assert(sizeof(number_type<5>) == 8);
static_assert(sizeof(number_type<6>) == 8);
static_assert(sizeof(number_type<7>) == 8);
static_assert(sizeof(number_type<8>) == 8);
static_assert(sizeof(number_type<9>) == 8);
```

- In a type template parameter, to select between integer and real uniform distribution, depending on whether the type template parameter is of an integral or floating point type:

```
template <typename T,
          typename D = std::conditional_t<
                         std::is_integral<T>::value,
                         std::uniform_int_distribution<T>,
                         std::uniform_real_distribution<T>>,
          typename = typename std::enable_if<
                         std::is_arithmetic<T>::value>::type>
std::vector<T> GenerateRandom(T const min, T const max,
                              size_t const size)
{
  std::vector<T> v(size);
```

```
  std::random_device rd{};
  std::mt19937 mt{ rd() };

  D dist{ min, max };

  std::generate(std::begin(v), std::end(v),
    [&dist, &mt] {return dist(mt); });

  return v;
}

auto v1 = GenerateRandom(1, 10, 10);     // integers
auto v2 = GenerateRandom(1.0, 10.0, 10); // doubles
```

How it works…

`std::conditional` is a class template that defines a member called `type` as either one or the other of its two type template parameters. The selection is done based on a compile-time constant Boolean expression provided as an argument for a non-type template parameter. Its implementation looks like this:

```
template<bool Test, class T1, class T2>
struct conditional
{
  typedef T2 type;
};

template<class T1, class T2>
struct conditional<true, T1, T2>
{
  typedef T1 type;
};
```

To help simplify the use of `std::conditional`, C++14 provides an alias template called `std::conditional_t`, that we have seen in the third example above, and that is defined as follows:

```
template<bool Test, class T1, class T2>
using conditional_t = typename conditional<Test,T1,T2>::type;
```

Let’s summarize the examples from the previous section:

- In the first example, if the platform is 32-bit, then the size of the pointer type is 4 bytes and, therefore, the compile-time expression `sizeof(void*) <= 4` is `true`; as a result, `std::conditional` defines its member type as `long`. If the platform is 64-bit, then the condition evaluates to `false`, because the size of the pointer type is 8 bytes, and therefore the member type is defined as `long long`.
- A similar situation is encountered in the second example, where `std::conditional` is used multiple times to emulate a series of `if...else` statements to select an appropriate type.
- In the third example, we used the alias template `std::conditional_t` to simplify the declaration of the function template `GenerateRandom`. Here, `std::conditional` is used to define the default value for a type template parameter representing a statistical distribution. Depending on whether the first type template parameter `T` is an integral or floating point type, the default distribution type is chosen between `std::uniform_int_distribution<T>` and `std::uniform_real_distribution<T>`. Use of other types is disabled by employing `std::enable_if` with a third template parameter, as we have seen in other recipes before.

See also

- *Using type traits to query properties of types*
- *Writing your own type traits*
- *Conditionally compiling classes and functions with enable_if* recipe of Chapter 4, Preprocessor and compilation

Working with Files and Streams

The recipes available in this chapter are as follows:

- Reading and writing raw data from/to binary files
- Reading and writing objects from/to binary files
- Using localized settings for streams
- Using I/O manipulators to control the output of a stream
- Using monetary I/O manipulators
- Using time I/O manipulators
- Working with filesystem paths
- Creating, copying, and deleting files and directories
- Removing content from a file
- Checking the properties of an existing file or directory
- Enumerating the content of a directory
- Finding a file

Introduction

One of the most important parts of the C++ standard library is the input/output, stream-based library that enables developers to work with files, memory streams, or other types of I/O devices. The first part of the chapter provides solutions to some common stream operations, such as reading and writing data, localization settings, and manipulating the input and output of a stream. The second part of the chapter explores the new C++17 `filesystem` library that enables developers to perform operations with the filesystem and its objects, such as files and directories.

Reading and writing raw data from/to binary files

Some of the data programs work with has to be persisted to disk files in various ways, that can include storing it in a database or to flat files, either as text or binary data. This recipe and the next one are focused on persisting and loading both raw data and objects from and to binary files. In this context, raw data means unstructured data, and in this recipe, we will consider writing and reading the content of a buffer (that is, a contiguous sequence of memory, that can be either a C-like array, an `std::vector`, or an `std::array`).

Getting ready

For this recipe, you should be familiar with the standard stream input/output library, though some explanations, to the extent required to understand this recipe, are provided below. You should also be familiar with the difference between binary and text files.

In this recipe, we will use the `ofstream` and `ifstream` classes, available in the namespace `std` in the `<fstream>` header.

In the following examples, we will consider the following data to write to a binary file (and consequently to read back):

```
    std::vector<unsigned char> output {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

How to do it…

To write the content of a buffer (in our example, an `std::vector`) to a binary file, you should perform the following steps:

1. Open a file stream for writing in binary mode by creating an instance of the `std::ofstream` class:

```
std::ofstream ofile("sample.bin", std::ios::binary);
```

2. Ensure that the file is actually open before writing data to the file:

```
if(ofile.is_open())
{
  // streamed file operations
}
```

3. Write the data to the file by providing a pointer to the array of characters and the number of characters to write:

```
ofile.write(reinterpret_cast<char*>(output.data()),
            output.size());
```

4. Flush the content of the stream buffer to the actual disk file; this is automatically done when you close the stream:

```
ofile.close();
```

To read the entire content of a binary file to a buffer, you should perform the following steps:

1. Open a file stream to read from a file in the binary mode by creating an instance of the `std::ifstream` class:

```
std::ifstream ifile("sample.bin", std::ios::binary);
```

2. Ensure that the file is actually opened before reading data from it:

```
if(ifile.is_open())
{
  // streamed file operations
}
```

3. Determine the length of the file by positioning the input position indicator to the end of the file, read its value, and then move the indicator to the beginning:

```
ifile.seekg(0, std::ios_base::end);
auto length = ifile.tellg();
ifile.seekg(0, std::ios_base::beg);
```

4. Allocate memory to read the content of the file:

```
std::vector<unsigned char> input;
input.resize(static_cast<size_t>(length));
```

5. Read the content of the file to the allocated buffer by providing a pointer to the array of characters for receiving the data and the number of characters to read:

```
        ifile.read(reinterpret_cast<char*>(input.data()), length);
```

6. Check that the read operation is completed successfully:

```
        auto success = !ifile.fail() && length == ifile.gcount();
```

7. Finally, close the file stream:

```
        ifile.close();
```

How it works…

The standard stream-based input/output library provides various classes that implement high-level input, output, or both input and output file stream, string stream and character array operations, manipulators that control how these streams behave, and several predefined stream objects (`cin`/`wcin`, `cout`/`wcout`, `cerr`/`wcerr`, and `clog`/`wclog`).

These streams are implemented as class templates and, for files, the library provides several classes:

- `basic_filebuf` implements the input/output operations for a raw file and is similar in semantics with a C `FILE` stream.
- `basic_ifstream` implements the high-level file stream input operations defined by the `basic_istream` stream interface, internally using a `basic_filebuf` object.
- `basic_ofstream` implements the high-level file stream output operations defined by the `basic_ostream` stream interface, internally using a `basic_filebuf` object.
- `basic_fstream` implements the high-level file stream input and output operations defined by the `basic_iostream` stream interface, internally using a `basic_filebuf` object.

Several typedefs for the class templates mentioned in the preceding classes are also defined in the `<fstream>` header. The `ofstream` and `ifstream` objects are the type synonyms used in the preceding examples:

```
typedef basic_ifstream<char>    ifstream;
typedef basic_ifstream<wchar_t> wifstream;
typedef basic_ofstream<char>    ofstream;
typedef basic_ofstream<wchar_t> wofstream;
typedef basic_fstream<char>     fstream;
typedef basic_fstream<wchar_t>  wfstream;
```

In the previous section, we saw how we can write and read raw data to and from a file stream. The way that works is explained in more detail here:

- To write data to a file, we instantiated an object of the type `std::ofstream`. In the constructor, we passed the name of the file to be opened and the stream open mode, for which we specified `std::ios::binary` to indicate binary mode. Opening the file like this discards the previous file content. If you want to append content to an existing file, you should also use the flag `std::ios::app` (that is, `std::ios::app | std::ios::binary`). This constructor internally calls `open()` on its underlying raw file object, that is, a `basic_filebuf` object. If this operation fails, a fail bit is set. To check whether the stream has been successfully associated with a file device, we used `is_open()` (that internally calls the method with the same name from the underlying `basic_filebuf`). Writing data to the file stream is done with the `write()` method that takes a pointer to the string of characters to write and the number of characters to write. Since this method operates with strings of characters, a `reinterpret_cast` is necessary if data is of another type, such as `unsigned char` in our example. The write operation does not set a fail bit on failure, but may throw an `std::ios_base::failure` exception. However, data is not written directly to the file device, but stored in the `basic_filebuf` object. To write it to the file,

the buffer needs to be flushed, which is done by calling `flush()`. This is done automatically when closing the file stream, as in the preceding example.

- To read data from a file, we instantiated an object of time `std::ifstream`. In the constructor, we passed the same arguments we used for opening the file for writing, the name of the file and the open mode, that is, `std::ios::binary`. The constructor internally calls `open()` on the underlying `std::basic_filebuf` object. To check whether the stream has been successfully associated with a file device, we used `is_open()` (that internally calls the method with the same name from the underlying `basic_filebuf`). In this example, we read the entire content of the file to a memory buffer, in particular, an `std::vector`. Before we can read the data, we must know the size of the file in order to allocate a buffer large enough to hold that data. To do so, we used `seekg()` to move the input position indicator to the end of the file, then we called `tellg()` to return the current position, which in this case indicates the size of the file, in bytes, and then we moved the input position indicator to the beginning of the file to be able to start reading from the beginning. Calling `seekg()` to move the position indicator to the end can be avoided by opening the file with the position indicator moved directly to the end. This can be achieved using the `std::ios::ate` opening flag in the constructor (or the `open()` method). After allocating enough memory for the content of the file, we copied the data from the file into memory using the `read()` method. This takes a pointer to the string of characters that receives the data read from the stream and the number of characters to be read. Since the stream operates on characters, a `reinterpret_cast` expression is necessary if the buffer contains other types of data, such as `unsigned char` in our example. This operation throws an `std::basic_ios::failure` exception if an error occurs. To determine the number of characters that have been successfully read from the stream, we can use the `gcount()` method. Upon completing the read operation, we close the file stream.

The operations shown in these examples are the minimal ones necessary to write and read data to and from file streams. It is important, though, that you perform appropriate checks for the success of the operations and catch possible exceptions that could occur.

The example code discussed so far in this recipe can be reorganized in the form of two general functions for writing and reading data to and from a file:

```
bool write_data(char const * const filename,
                char const * const data,
                size_t const size)
{
  auto success = false;
  std::ofstream ofile(filename, std::ios::binary);

  if(ofile.is_open())
  {
    try
    {
      ofile.write(data, size);
       success = true;
    }
    catch(std::ios_base::failure &)
    {
      // handle the error
    }
    ofile.close();
```

```
  }

  return success;
}

size_t read_data(char const * const filename,
                 std::function<char*(size_t const)> allocator)
{
  size_t readbytes = 0;
  std::ifstream ifile(filename, std::ios::ate | std::ios::binary);

  if(ifile.is_open())
  {
    auto length = static_cast<size_t>(ifile.tellg());
    ifile.seekg(0, std::ios_base::beg);

    auto buffer = allocator(length);

    try
    {
      ifile.read(buffer, length);

      readbytes = static_cast<size_t>(ifile.gcount());
    }
    catch (std::ios_base::failure &)
    {
      // handle the error
    }

    ifile.close();
  }

  return readbytes;
}
```

`write_data()` is a function that takes the name of a file and a pointer to an array of character and its length and writes it to the specified file. `read_data()` is a function that takes the name of a file and a function that allocates a buffer and reads the entire content of the file to the buffer returned by the allocated function. The following is an example of how these functions can be used:

```
std::vector<unsigned char> output {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<unsigned char> input;

if(write_data("sample.bin",
              reinterpret_cast<char*>(output.data()),
              output.size()))
{
  if(read_data("sample.bin",
               [&input](size_t const length) {
    input.resize(length);
    return reinterpret_cast<char*>(input.data());}) > 0)
    {
      std::cout << (output == input ? "equal": "not equal")
                << std::endl;
    }
  }
```

Alternatively, we could use a dynamically allocated buffer, instead of the `std::vector`; the changes required for that are small in the overall example:

```
std::vector<unsigned char> output {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
unsigned char* input = nullptr;
size_t readb = 0;

if(write_data("sample.bin",
              reinterpret_cast<char*>(output.data()),
              output.size()))
{
  if((readb = read_data(
     "sample.bin",
     [&input](size_t const length) {
```

```
        input = new unsigned char[length];
        return reinterpret_cast<char*>(input); })) > 0)
        {
          auto cmp = memcmp(output.data(), input, output.size());
          std::cout << (cmp == 0 ? "equal": "not equal")
                    << std::endl;
        }
      }

      delete [] input;
```

There’s more…

The way of reading data from a file to memory shown in this recipe is only one of the several possible alternatives. Compared to the others, it is, however, the fastest method, even though the alternatives may look more appealing from an object-oriented perspective. It is beyond the purpose of this recipe to compare the performance of these alternatives, but the reader can take it as an exercise.

The following are possible alternatives for reading data from a file stream:

- Initializing an `std::vector` directly using `std::istreambuf_iterator` iterators (similarly, this can be used with `std::string`):

```
std::vector<unsigned char> input;
std::ifstream ifile("sample.bin", std::ios::binary);
if(ifile.is_open())
{
  input = std::vector<unsigned char>(
     std::istreambuf_iterator<char>(ifile),
     std::istreambuf_iterator<char>());
  ifile.close();
}
```

- Assigning the content of an `std::vector` from `std::istreambuf_iterator` iterators:

```
std::vector<unsigned char> input;
std::ifstream ifile("sample.bin", std::ios::binary);
if(ifile.is_open())
{
  ifile.seekg(0, std::ios_base::end);
  auto length = ifile.tellg();
  ifile.seekg(0, std::ios_base::beg);

  input.reserve(static_cast<size_t>(length));
  input.assign(
    std::istreambuf_iterator<char>(ifile),
    std::istreambuf_iterator<char>());
  ifile.close();
}
```

- Copying the content of the file stream to a vector using `std::istreambuf_iterator` iterators and an `std::back_inserter` adapter to write to the end of the vector:

```
std::vector<unsigned char> input;
std::ifstream ifile("sample.bin", std::ios::binary);
if(ifile.is_open())
{
  ifile.seekg(0, std::ios_base::end);
  auto length = ifile.tellg();
  ifile.seekg(0, std::ios_base::beg);

  input.reserve(static_cast<size_t>(length));
  std::copy(std::istreambuf_iterator<char>(ifile),
            std::istreambuf_iterator<char>(),
            std::back_inserter(input));
  ifile.close();
}
```

See also

- *Reading and writing objects from/to binary files*
- *Using I/O manipulators to control the output of a stream*

Reading and writing objects from/to binary files

In the previous recipe, we saw how to write and read raw data (that is, unstructured data) to and from a file. Many times, however, we have to persist and load objects. Writing and reading in the manner shown in the previous recipe works for POD types only. For anything else, we must explicitly decide what is actually written or read, as writing or reading pointers, virtual tables, and any sort of meta data is not only irrelevant, but also semantically wrong. These operations are commonly referred to as serialization and deserialization. In this recipe, we will see how we can serialize and deserialize both POD and non-POD types to and from binary files.

Getting ready

It is recommended that you first read the previous recipe, *Reading and writing raw data from/to binary files*, before you continue. You should also know what POD and non-POD types are and how operators can be overloaded.

For the examples in this recipe, we will use the `foo` and `foopod` classes shown in the following:

```
class foo
{
  int i;
  char c;
  std::string s;

public:
  foo(int const i = 0, char const c = 0, std::string const & s = {}):
    i(i), c(c), s(s)
  {}

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

  bool operator==(foo const & rhv) const
  {
    return i == rhv.i &&
           c == rhv.c &&
           s == rhv.s;
  }

      bool operator!=(foo const & rhv) const
  {
    return !(*this == rhv);
  }
};

struct foopod
{
  bool a;
  char b;
  int c[2];
};
```

```


bool operator==(foopod const & f1, foopod const & f2)
{
  return f1.a == f2.a && f1.b == f2.b &&
         f1.c[0] == f2.c[0] && f1.c[1] == f2.c[1];
}
```

How to do it…

To serialize/deserialize POD types that do not contain pointers, use `ofstream::write()` and `ifstream::read()`, as shown in the previous recipe:

- Serialize objects to a binary file using `ofstream` and the `write()` method:

```
std::vector<foopod> output {
  {true, '1', {1, 2}},
  {true, '2', {3, 4}},
  {false, '3', {4, 5}}
};

std::ofstream ofile("sample.bin", std::ios::binary);
if(ofile.is_open())
{
  for(auto const & value : output)
  {
    ofile.write(reinterpret_cast<const char*>(&value),
                sizeof(value));
  }

  ofile.close();
}
```

- Deserialize objects from a binary file using the `ifstream` and `read()` methods:

```
std::vector<foopod> input;
std::ifstream ifile("sample.bin", std::ios::binary);
if(ifile.is_open())
{
  while(true)
  {
    foopod value;
    ifile.read(reinterpret_cast<char*>(&value),
               sizeof(value));

    if(ifile.fail() || ifile.eof()) break;
      input.push_back(value);
  }

  ifile.close();
}
```

To serialize non-POD types (or POD types that contain pointers), you must explicitly write the value of data members to a file, and to deserialize, you must explicitly read from the file to the data members in the same order. To exemplify this, we will consider the `foo` class defined earlier:

- Add a member function called `write()` to serialize objects of this class. The method takes a reference to an `ofstream` and returns a `bool` indicating whether the operation was successful or not:

```
bool write(std::ofstream& ofile) const
{
  ofile.write(reinterpret_cast<const char*>(&i), sizeof(i));
  ofile.write(&c, sizeof(c));
  auto size = static_cast<int>(s.size());
  ofile.write(reinterpret_cast<char*>(&size), sizeof(size));
  ofile.write(s.data(), s.size());

  return !ofile.fail();
```

```
}
```

- Add a member function called `read()` to deserialize objects of this class. This method takes a reference to an `ifstream` and returns a `bool` indicating whether the operation was successful or not:

```
bool read(std::ifstream& ifile)
{
  ifile.read(reinterpret_cast<char*>(&i), sizeof(i));
  ifile.read(&c, sizeof(c));
  auto size {0};
  ifile.read(reinterpret_cast<char*>(&size), sizeof(size));
  s.resize(size);
  ifile.read(reinterpret_cast<char*>(&s.front()), size);

  return !ifile.fail();
}
```

An alternative to the `write()` and `read()` member functions exemplified above is to overload `operator<<` and `operator>>`. To do this, you should perform the following steps:

1. Add `friend` declarations for non-member `operator<<` and `operator>>` to the class to be serialized/deserialized (in this case, the `foo` class):

```
friend std::ofstream& operator<<(std::ofstream& ofile,
                                 foo const& f);
friend std::ifstream& operator>>(std::ifstream& ifile,
                                 foo& f);
```

2. Overload `operator<<` for your class:

```
std::ofstream& operator<<(std::ofstream& ofile, foo const& f)
{
  ofile.write(reinterpret_cast<const char*>(&f.i),
              sizeof(f.i));
  ofile.write(&f.c, sizeof(f.c));
  auto size = static_cast<int>(f.s.size());
  ofile.write(reinterpret_cast<char*>(&size), sizeof(size));
  ofile.write(f.s.data(), f.s.size());

  return ofile;
}
```

3. Overload `operator>>` for your class:

```
std::ifstream& operator>>(std::ifstream& ifile, foo& f)
{
  ifile.read(reinterpret_cast<char*>(&f.i), sizeof(f.i));
  ifile.read(&f.c, sizeof(f.c));
  auto size {0};
  ifile.read(reinterpret_cast<char*>(&size), sizeof(size));
  f.s.resize(size);
  ifile.read(reinterpret_cast<char*>(&f.s.front()), size);

  return ifile;
}
```

How it works…

Regardless of whether we serialize the entire object (for POD types) or only parts of it, we use the same stream classes discussed in the previous recipe, `ofstream` for output file streams and `ifstream` for input file streams. Details about writing and reading data using these standard classes have been discussed in that recipe and will not be reiterated here.

When you serialize and deserialize objects to and from files, you should avoid writing values of pointers to a file, and you must not read pointer values from the file since these represent memory addresses and are meaningless across processes, or even in the same process some moments later. Instead, you should write data referred by a pointer and read data into objects referred by a pointer. This is a general principle, and in practice, you may encounter situations where a source may have multiple pointers to the same object, in which case you might want to write only one copy and also handle the reading in a corresponding manner.

If the objects you want to serialize are of a POD type, you can do it just like we did when we discussed raw data. In the example in this recipe, we serialized a sequence of objects of the `foopod` type. When we deserialize, we read from the file stream in a loop until the end of the file is read or a failure occurs. The way reading is done in this case may look counter-intuitive, but doing it differently may lead to duplication of the last read value:

- Reading is done in an infinite loop.
- A read operation is performed in the loop.
- A check for failure or end of file is performed, and if any of these occurred, the infinite loop is exited.
- The value is added to the input sequence and the looping continues.

If reading is done using a loop with an exit condition that checks the end of the file bit, that is, `while(!ifile.eof())`, the last value will be added twice to the input sequence. The reason for that, is that upon reading the last value, the end of file is not yet encountered (as that is a mark beyond the last byte of the file). The end of file mark is only reached at the next read attempt, which, therefore, sets the of bit of the stream. However, the input variable still has the last value, as it hasn't been overwritten with anything, and this is added for a second time to the input vector.

If the objects you want to serialize and deserialize are of non-POD types, writing/reading these objects as raw data is not possible. For instance, such an object may have a virtual table. Writing the vtable to a file does not cause problems, even though it does not have any value; however, reading from a file, and, therefore, overwriting the vtable of an object will have catastrophic effects on the object and the program.

When serializing/deserializing non-POD types, there are various alternatives, some of them shown in the previous section: either provide explicit methods for writing and reading or overloading the standard `<<` and `>>` operators. The second approach has the advantage that it enables the use of your class in generic code where objects are written

and read to and from stream files using these operators.

When you plan to serialize and deserialize your objects, consider versioning your data from the very beginning to avoid problems if the structure of your data changes over time. How versioning should be done is beyond the purpose of this recipe.

See also

- *Reading and writing raw data from/to binary files*
- *Using I/O manipulators to control the output of a stream*

Using localized settings for streams

The way writing or reading to and from streams is performed may depend on language and regional settings. Examples include writing and parsing numbers, time values, or monetary values, or comparing (collating) strings. The C++ input/output library provides a general purpose mechanism for handling internationalization features through *locales* and *facets*. In this recipe, you will learn how to use locales to control the behavior of input/output streams.

Getting ready

All the examples in this recipe are using the `std::cout` predefined console stream object. However, the same applies to all input/output stream objects. Also, in this recipe examples, we will use the following objects and lambda function:

```
auto now = std::chrono::system_clock::now();
auto stime = std::chrono::system_clock::to_time_t(now);
auto ltime = std::localtime(&stime);

std::vector<std::string> names
  {"John", "adele", "Øivind", "François", "Robert", "Åke"};

auto sort_and_print = [](std::vector<std::string> v,
                         std::locale const & loc)
{
  std::sort(v.begin(), v.end(), loc);
  for (auto const & s : v) std::cout << s << ' ';
  std::cout << std::endl;
};
```

The locale names used in this recipe (*en_US.utf8*, *de_DE.utf8*, and so on) are the ones used on UNIX systems. The following table lists their equivalent for Windows systems:

UNIX	**Windows**
en_US.utf8	English_US.1252
en_GB.utf8	English_UK.1252
de_DE.utf8	German_Germany.1252
sv_SE.utf8	Swedish_Sweden.1252

How to do it…

To control the localization settings of a stream, you must do the following:

- Use `std::locale` class to represent the localization settings. There are various ways to construct locale objects including the following:
  - Default construct it to use the global locale (by default, the *C* locale at the program startup).
  - From a local name, such as *C*, *POSIX*, *en_US.utf8*, and so on, if supported by the operating system.
  - From another locale, except for a specified facet.
  - From another locale, except for all the facets from a specified category that are copied from another specified locale:

```
// default construct
auto loc_def = std::locale {};

// from a name
auto loc_us = std::locale {"en_US.utf8"};

// from another locale except for a facet
auto loc1 = std::locale {loc_def, new std::collate<wchar_t>};

// from another local, except the facet in a category
auto loc2 = std::locale {loc_def, loc_us,
                         std::locale::collate};
```

- To get a copy of the default *C* locale, use the `std::locale::classic()` static method:

```
auto loc = std::locale::classic();
```

- To change the default locale that is copied every time a locale is default constructed, use the `std::locale::global()` static method:

```
std::locale::global(std::locale("en_US.utf8"));
```

- Use the `imbue()` method to change the current locale of an input/output stream:

```
std::cout.imbue(std::locale("en_US.utf8"));
```

The following list shows examples of using various locales:

- Use a particular locale, indicated by its name. In this example, the locale is for German:

```
auto loc = std::locale("de_DE.utf8");
std::cout.imbue(loc);

std::cout << 1000.50 << std::endl;
// 1.000,5
std::cout << std::showbase << std::put_money(1050)
          << std::endl;
// 10,50 €
std::cout << std::put_time(ltime, "%c") << std::endl;
// So 04 Dez 2016 17:54:06 JST
sort_and_print(names, loc);
```

```
        // adele Åke François John Øivind Robert
```

- Use a locale corresponding to the user settings (as defined in the system). This is done by constructing an `std::locale` object from an empty string:

```
        auto loc = std::locale("");
        std::cout.imbue(loc);

        std::cout << 1000.50 << std::endl;
        // 1,000.5
        std::cout << std::showbase << std::put_money(1050)
                  << std::endl;
        // $10.50
        std::cout << std::put_time(ltime, "%c") << std::endl;
        // Sun 04 Dec 2016 05:54:06 PM JST
        sort_and_print(names, loc);
        // adele Åke François John Øivind Robert
```

- Set and use the global locale:

```
        std::locale::global(std::locale("sv_SE.utf8")); // set global
        auto loc = std::locale{};                       // use global
        std::cout.imbue(loc);

        std::cout << 1000.50 << std::endl;
        // 1 000,5
        std::cout << std::showbase << std::put_money(1050)
                  << std::endl;
        // 10,50 kr
        std::cout << std::put_time(ltime, "%c") << std::endl;
        // sön 4 dec 2016 18:02:29
        sort_and_print(names, loc);
        // adele François John Robert Åke Øivind
```

- Use the default *C* locale:

```
        auto loc = std::locale::classic();
        std::cout.imbue(loc);

        std::cout << 1000.50 << std::endl;
        // 1000.5
        std::cout << std::showbase << std::put_money(1050)
                  << std::endl;
        // 1050
        std::cout << std::put_time(ltime, "%c") << std::endl;
        // Sun Dec 4 17:55:14 2016
        sort_and_print(names, loc);
        // François John Robert adele Åke Øivind
```

How it works…

A locale object does not actually store localized settings. A *locale* is a heterogeneous container of facets. A *facet* is an object that defines localization and internationalization settings. The standard defines a list of facets that each locale must contain. In addition to this, a locale can contain any other user-defined facets. The following is a list of all standard-defined facets:

`std::collate<char>`	`std::collate<wchar_t>`
`std::ctype<char>`	`std::ctype<wchar_t>`
`std::codecvt<char,char,mbstate_t>` `std::codecvt<char16_t,char,mbstate_t>`	`std::codecvt<char32_t,char,mbstate_t>` `std::codecvt<wchar_t,char,mbstate_t>`
`std::moneypunct<char>` `std::moneypunct<char,true>`	`std::moneypunct<wchar_t>` `std::moneypunct<wchar_t,true>`
`std::money_get<char>`	`std::money_get<wchar_t>`
`std::money_put<char>`	`std::money_put<wchar_t>`
`std::numpunct<char>`	`std::numpunct<wchar_t>`
`std::num_get<char>`	`std::num_get<wchar_t>`
`std::num_put<char>`	`std::num_put<wchar_t>`
`std::time_get<char>`	`std::time_get<wchar_t>`
`std::time_put<char>`	`std::time_put<wchar_t>`
`std::messages<char>`	`std::messages<wchar_t>`

A locale is an immutable object containing immutable facet objects. Locales are implemented as a reference-counted array of reference-counted pointers to facets. The array is indexed by `std::locale::id` and all facets must be derived from the base class `std::locale::facet` and must have a public static member of the `std::locale::id` type called `id`.

It is only possible to create a locale object using one of the overloaded constructors or with the `combine()` method that, as the name implies, combines the current locale with a new compile-time identifiable facet and returns a new locale object. On the other hand, it is possible to determine whether a locale contains a particular facet using the `std::has_facet()` function template, or to obtain a reference to a facet implemented by a particular locale, using the `std::use_facet()` function template.

In the preceding examples, we have sorted a vector of strings and passed a locale object as the third argument to the `std::sort()` general algorithm. This third argument is supposed to be a comparison function object. Passing a locale object works because `std::locale` has an `operator()` that lexicographically compares two strings using its collate facet. This is actually the only localization functionality directly provided by `std::locale`; however, what this does is invoke the collate facet's `compare()` method that performs the string comparison based on the facet's rules.

Every program has a global locale created when the program starts. The content of this global locale is copied into every default constructed locale. The global locale can be replaced using the static method `std::locale::global()`. By default, the global locale is the

C locale, a locale equivalent to ANCI C's locale with the same name. This locale was created for handling simple English texts, and it is the default one in C++ to provide compatibility with C. A reference to this locale can be obtained with the static method `std::locale::classic()`.

By default, all streams use the classic locale to write or parse text. However, it is possible to change the locale used by a stream using the stream's `imbue()` method. This is a member of the `std::ios_base` class that is the base for all input/output streams. A companion member is the `getloc()` method that returns a copy of the current stream's locale.

In the preceding examples, we changed the locale for the `std::cout` stream object. In practice, you may want to set the same locale for all stream objects associated with the standard C streams: `cin`, `cout`, `cerr`, and `clog` (or `wcin`, `wcout`, `wcerr`, and `wclog`).

See also

- *Using I/O manipulators to control the output of a stream*
- *Using monetary I/O manipulators*
- *Using time I/O manipulators*

Using I/O manipulators to control the output of a stream

Apart from the stream-based input/output library, the standard library provides a series of helper functions, called manipulators, that control the input or output streams using `operator<<` and `operator>>`. In this recipe, we will look at some of these manipulators and demonstrate their use through some examples that format the output to the console, and will continue showing more manipulators in the next recipes.

Getting ready

The I/O manipulators are available in the `std` namespace in headers `<ios>`, `<istream>`, `<ostream>`, and `<iomanip>`. In this recipe, we will only discuss some of the manipulators from `<ios>` and `<iomanip>`.

How to do it…

The following manipulators can be used to control the output or input of a stream:

- `boolalpha` and `noboolalpha` enable and disable textual representation of Booleans:

```
std::cout << std::boolalpha << true << std::endl;    // true
std::cout << false << std::endl;                     // false
std::cout << std::noboolalpha << false << std::endl; // 0
```

- `left`, `right`, and `internal` affect the alignment of the fill characters; `left` and `right` affect all text, but `internal` affects only integer, floating point, and monetary output:

```
std::cout << std::right << std::setw(10) << "right"
          << std::endl;
std::cout << std::setw(10) << "text" << std::endl;
std::cout << std::left << std::setw(10) << "left" << std::endl;
```

- `fixed`, `scientific`, `hexfloat`, and `defaultfloat` change the formatting used for floating-point types (for both input and output streams). The last two are available only since C++11:

```
std::cout << std::fixed << 0.25 << std::endl;
// 0.250000
std::cout << std::scientific << 0.25 << std::endl;
// 2.500000e-01
std::cout << std::hexfloat << 0.25 << std::endl;
// 0x1p-2
std::cout << std::defaultfloat << 0.25 << std::endl;
// 0.25
```

- `dec`, `hex`, and `oct` control the base used for integer types (both in input and output streams):

```
std::cout << std::oct << 42 << std::endl; // 52
std::cout << std::hex << 42 << std::endl; // 2a
std::cout << std::dec << 42 << std::endl; // 42
```

- `setw` changes the width of the next input or output field. The default width is 0.
- `setfill` changes the fill character for the output stream; this is the character that is used to fill the next fields until the specified width is reached. The default fill character is whitespace:

```
std::cout << std::right
          << std::setfill('.') << std::setw(10)
          << "right" << std::endl;
// .....right
```

- `setprecision` changes the decimal precision (how many digits are generated) for the floating-point types in both input and output streams. The default precision is 6:

```
std::cout << std::fixed << std::setprecision(2) << 12.345
          << std::endl;
// 12.35
```

How it works…

All of the I/O manipulators listed above, with the exception of `setw`, that only refers to the next output field, affect the stream and all consecutive writing or reading operations use the last specified format until another manipulator is used again.

Some of these manipulators are called without arguments. Examples include `boolalpha`/`noboolalpha` or `dec`/`hex`/`oct`. These manipulators are functions that take a single argument, that is a reference to a string, and return a reference to the same stream:

```
std::ios_base& hex(std::ios_base& str);
```

Expressions, such as `std::cout << std::hex`, are possible because both `basic_ostream::operator<<` and `basic_istream::operator>>` have special overloads that take a pointer to these functions.

Other manipulators, including some that are not mentioned here, are invoked with arguments. These manipulators are functions that take one or more arguments and return an object of an unspecified type:

```
template<class CharT>
/*unspecified*/ setfill(CharT c);
```

To better exemplify the use of these manipulators, we will consider two examples that format output to the console.

In the first example, we will list the table of contents of a book with the following requirements:

- The chapter number is right-aligned and shown with Roman numerals.
- The chapter title is left-aligned and the remaining space until the page number is filled with dots.
- The page number of the chapter is right-aligned.

For this example, we will use the following classes and helper function:

```
struct Chapter
{
  int Number;
  std::string Title;
  int Page;
};

struct BookPart
{
  std::string Title;
  std::vector<Chapter> Chapters;
};

struct Book
{
  std::string Title;
  std::vector<BookPart> Parts;
};

std::string to_roman(unsigned int value)
{
  struct roman_t { unsigned int value; char const* numeral; };
  const static roman_t rarr[13] =
  {
```

```
      {1000, "M"}, {900, "CM"}, {500, "D"}, {400, "CD"},
      {100, "C"}, { 90, "XC"}, { 50, "L"}, { 40, "XL"},
      { 10, "X"}, { 9, "IX"}, { 5, "V"}, { 4, "IV"},
      { 1, "I"}
    };

    std::string result;
    for (auto const & number : rarr)
    {
      while (value >= number.value)
      {
        result += number.numeral;
        value -= number.value;
      }
    }

    return result;
  }
```

The `print_toc()` function, shown in the following code snippet, takes a `Book` as its argument and prints its content to the console, according to the specified requirements. For this purpose, we use the following:

- `std::left` and `std::right` specify the text alignment.
- `std::setw` specifies the width of each output field.
- `std::fill` specifies the fill character (a space for the chapter number, and a dot for the chapter title):

```
  void print_toc(Book const & book)
  {
    std::cout << book.Title << std::endl;
    for(auto const & part : book.Parts)
    {
      std::cout << std::left << std::setw(15) << std::setfill(' ')
                << part.Title << std::endl;
      std::cout << std::left << std::setw(15) << std::setfill('-')
                << '-' << std::endl;

      for(auto const & chapter : part.Chapters)
      {
        std::cout << std::right << std::setw(4) << std::setfill(' ')
                  << to_roman(chapter.Number) << ' ';
        std::cout << std::left << std::setw(35) << std::setfill('.')
                  << chapter.Title;
        std::cout << std::right << std::setw(3) << std::setfill('.')
                  << chapter.Page << std::endl;
      }
    }
  }
```

The following is an example of using this method, with a `Book` object describing the table of contents from the book *The Fellowship of the Ring*:

```
  auto book = Book
  {
    "THE FELLOWSHIP OF THE RING"s,
    {
      {
        "BOOK ONE"s,
        {
          {1, "A Long-expected Party"s, 21},
          {2, "The Shadow of the Past"s, 42},
          {3, "Three Is Company"s, 65},
          {4, "A Short Cut to Mushrooms"s, 86},
          {5, "A Conspiracy Unmasked"s, 98},
          {6, "The Old Forest"s, 109},
          {7, "In the House of Tom Bombadil"s, 123},
          {8, "Fog on the Barrow-downs"s, 135},
          {9, "At the Sign of The Prancing Pony"s, 149},
```

```
            {10, "Strider"s, 163},
            {11, "A Knife in the Dark"s, 176},
            {12, "Flight to the Ford"s, 197},
          },
        },
        {
          "BOOK TWO"s,
          {
            {1, "Many Meetings"s, 219},
            {2, "The Council of Elrond"s, 239},
            {3, "The Ring Goes South"s, 272},
            {4, "A Journey in the Dark"s, 295},
            {5, "The Bridge of Khazad-dum"s, 321},
            {6, "Lothlorien"s, 333},
            {7, "The Mirror of Galadriel"s, 353},
            {8, "Farewell to Lorien"s, 367},
            {9, "The Great River"s, 380},
            {10, "The Breaking of the Fellowship"s, 390},
          },
        },
      }
    };

    print_toc(book);
```

In this case, the output is the following:

```
THE FELLOWSHIP OF THE RING
BOOK ONE
---------------
   I A Long-expected Party.............21
  II The Shadow of the Past............42
 III Three Is Company..................65
  IV A Short Cut to Mushrooms..........86
   V A Conspiracy Unmasked............98
  VI The Old Forest....................109
 VII In the House of Tom Bombadil.....123
VIII Fog on the Barrow-downs...........135
  IX At the Sign of The Prancing Pony...149
   X Strider............................163
  XI A Knife in the Dark...............176
 XII Flight to the Ford.................197
BOOK TWO
---------------
   I Many Meetings.....................219
  II The Council of Elrond.............239
 III The Ring Goes South...............272
  IV A Journey in the Dark.............295
   V The Bridge of Khazad-dum..........321
  VI Lothlorien........................333
 VII The Mirror of Galadriel...........353
VIII Farewell to Lorien................367
  IX The Great River...................380
   X The Breaking of the Fellowship.....390
```

For the second example, the goal is to output a table with the largest companies in the world by revenue. The table will have columns for the company name, industry, revenue (in USD billions), increase/decrease of revenue growth, revenue growth, the number of employees, and country of origin. For this example, we will use the following class:

```
    struct Company
    {
      std::string Name;
      std::string Industry;
      double      Revenue;
      bool        RevenueIncrease;
      double      Growth;
      int         Employees;
      std::string Country;
    };
```

The `print_companies()` function in the following code snippet uses several additional

manipulators to the ones shown in the previous example:

- `std::boolalpha` displays Boolean values as `true` and `false` instead of `1` and `0`.
- `std::fixed` indicates a fixed floating-point representation, and then `std::defaultfloat` reverts to the default floating-point representation.
- `std::setprecision` specifies the number of decimal digits to be displayed in the output. Together with `std::fixed`, this is used to indicate a fixed representation with a decimal digit for the growth field.

```
void print_companies(std::vector<Company> const & companies)
{
  for(auto const & company : companies)
  {
    std::cout << std::left << std::setw(26) << std::setfill(' ')
              << company.Name;
    std::cout << std::left << std::setw(18) << std::setfill(' ')
              << company.Industry;
    std::cout << std::left << std::setw(5) << std::setfill(' ')
              << company.Revenue;
    std::cout << std::left << std::setw(5) << std::setfill(' ')
              << std::boolalpha << company.RevenueIncrease
              << std::noboolalpha;
    std::cout << std::right << std::setw(5) << std::setfill(' ')
              << std::fixed << std::setprecision(1) << company.Growth
              << std::defaultfloat << std::setprecision(6) << ' ';
    std::cout << std::right << std::setw(8) << std::setfill(' ')
              << company.Employees << ' ';
    std::cout << std::left << std::setw(2) << std::setfill(' ')
              << company.Country
              << std::endl;
  }
}
```

The following is an example of calling this method. The source of the data shown here is Wikipedia (https://en.wikipedia.org/wiki/List_of_largest_companies_by_revenue, as of 2016):

```
std::vector<Company> companies
{
  {"Walmart"s, "Retail"s, 482, false, 0.71,
   2300000, "US"s},
  {"State Grid"s, "Electric utility"s, 330, false, 2.91,
   927839, "China"s},
  {"Saudi Aramco"s, "Oil and gas"s, 311, true, 40.11,
   65266, "SA"s},
  {"China National Petroleum"s, "Oil and gas"s, 299,
   false, 30.21, 1589508, "China"s},
  {"Sinopec Group"s, "Oil and gas"s, 294, false, 34.11,
   810538, "China"s},
};

print_companies(companies);
```

In this case, the output has a table-based format, as follows; you can take it as an exercise to add a table heading and perhaps a grid line:

```
Walmart                  Retail           482  false  0.7  2300000 US
State Grid               Electric utility 330  false  2.9   927839 China
Saudi Aramco             Oil and gas      311  true  40.1    65266 SA
China National Petroleum Oil and gas      299  false 30.2  1589508 China
Sinopec Group            Oil and gas      294  false 34.1   810538 China
```

See also

- *Reading and writing raw data from/to binary files*
- *Using monetary I/O manipulators*
- *Using time I/O manipulators*

Using monetary I/O manipulators

In the previous recipe, we have looked at some of the manipulators that can be used to control the input and output streams. The manipulators discussed there were related to numeric and text values. In this recipe, we will see how to use standard manipulators to write and read monetary values.

Getting ready

You should be familiar with locales and how to set them for a stream. This topic is discussed in the *Using localized settings for streams* recipe. It is recommended that you read that recipe before continuing.

The manipulators discussed in this recipe are available in the `std` namespace in the `<iomanip>` header.

How to do it…

To write a monetary value to an output stream, you should do the following:

- Set the desired locale for controlling the monetary format:

```
std::cout.imbue(std::locale("en_GB.utf8"));
```

- Use either a `long double` or a `std::basic_string` value for the amount:

```
long double mon = 12345.67;
std::string smon = "12345.67";
```

- Use a `std::put_money` manipulator with a single argument, the monetary value, to display the value using the currency symbol (if any is available):

```
std::cout << std::showbase << std::put_money(mon)
          << std::endl; // £123.46
std::cout << std::showbase << std::put_money(smon)
          << std::endl; // £123.46
```

- Use `std::put_money` with two arguments, the monetary value and Boolean flag set to `true`, to indicate the use of an international currency string:

```
std::cout << std::showbase << std::put_money(mon, true)
          << std::endl; // GBP 123.46
std::cout << std::showbase << std::put_money(smon, true)
          << std::endl; // GBP 123.46
```

To read a monetary value from an input stream, you should do the following:

- Set the desired locale for controlling the monetary format:

```
std::istringstream stext("$123.45 123.45 USD");
stext.imbue(std::locale("en_US.utf8"));
```

- Use either a `long double` or `std::basic_string` value to read the amount from the input stream:

```
long double v1;
std::string v2;
```

- Use `std::get_money()` with a single argument, the variable where the monetary value is to be written, if a currency symbol may be used in the input stream:

```
stext >> std::get_money(v1) >> std::get_money(v2);
// v1 = 12345, v2 = "12345"
```

- Use `std::get_money()` with two arguments, the variable where the monetary value is to be written and a Boolean flag set to `true`, to indicate the presence of an international currency string:

```
stext >> std::get_money(v1, true) >> std::get_money(v2, true);
// v1 = 0, v2 = "12345"
```

How it works…

The `put_money()` and `get_money()` manipulators are very similar: they are both function templates that take an argument representing either the monetary value to be written to the output stream, or a variable to hold the monetary value read from an input stream, and a second, optional parameter, to indicate whether an international currency string is used. The default alternative is the currency symbol, if one is available. `put_money()` uses the `std::money_put()` facet settings to output a monetary value, and `get_money()` uses the `std::money_get()` facet to parse a monetary value. Both manipulator function templates return an object of an unspecified type. These functions do not throw exceptions:

```
template <class MoneyT>
/*unspecified*/ put_money(const MoneyT& mon, bool intl = false);

template <class MoneyT>
/*unspecified*/ get_money(MoneyT& mon, bool intl = false);
```

Both manipulator functions require the monetary value to be either a `long double` or a `std::basic_string`.

However, it is important to note that monetary values are stored as integral numbers of the smallest denomination of the currency defined by the locale in use. Considering US dollars as that currency, $100.00 is stored as 10000.0, and 1 cent, that is, $0.01, is stored as 1.0.

When writing a monetary value to an output stream, it is important to use the `std::showbase` manipulator if you want to display the currency symbol or the international currency string. This is normally used to indicate the prefix of a numeric base (such as 0x for hexadecimal), but for monetary values, it is used to indicate whether currency symbol/string should be displayed or not:

```
std::cout << std::put_money(12345.67)
          << std::endl; // prints 123.46
std::cout << std::showbase << std::put_money(12345.67)
          << std::endl; // prints £123.46
```

See also

- *Using I/O manipulators to control the output of a stream*
- *Using time I/O manipulators*

Using time I/O manipulators

Similar to the monetary input/output manipulators discussed in the previous recipe, the C++11 standard provides manipulators to control the writing and reading of time values to and from streams, time values represented in the form of an `std::tm` object that holds a calendar date and time. In this recipe, you will learn how to use these time manipulators.

Getting ready

Time values used by the time I/O manipulators are expressed in `std::tm` values. You should be familiar with this structure from the `<ctime>` header.

You should also be familiar with locales and how to set them for a stream. This topic is discussed in the *Using localized settings for streams* recipe. It is recommended that you read that recipe before continuing.

The manipulators discussed in this recipe are available in the `std` namespace in the `<iomanip>` header.

How to do it…

To write a time value to an output stream, you should perform the following steps:

1. Obtain a calendar date and time value corresponding to a given time. There are various ways to do that. The following are examples of converting the current time to a local time expressed as a calendar date and time:

```
auto now = std::chrono::system_clock::now();
auto stime = std::chrono::system_clock::to_time_t(now);
auto ltime = std::localtime(&stime);

auto ttime = std::time(nullptr);
auto ltime = std::localtime(&ttime);
```

2. Use `std::put_time()` supplying a pointer to the `std::tm` object representing the calendar date and time, and a pointer to a null-terminated character string representing the format. The C++11 standard provides a long list of formats that can be used; this list can be consulted at http://en.cppreference.com/w/cpp/io/manip/put_time.

3. To write a standard date and time string according to the settings of a specific locale, first set the locale for the stream by calling `imbue()` and then use the `std::put_time()` manipulator:

```
std::cout.imbue(std::locale("en_GB.utf8"));
std::cout << std::put_time(ltime, "%c") << std::endl;
// Sun 04 Dec 2016 05:26:47 JST
```

The following are examples of supported time formats:

- ISO 8601 date format `"%F"` or `"%Y-%m-%d"`:

```
std::cout << std::put_time(ltime, "%F") << std::endl;
// 2016-12-04
```

- ISO 8601 time format `"%T"`:

```
std::cout << std::put_time(ltime, "%T") << std::endl;
// 05:26:47
```

- ISO 8601 combined date and time in UTC format `"%FT%T%z"`:

```
std::cout << std::put_time(ltime, "%FT%T%z") << std::endl;
// 2016-12-04T05:26:47+0900
```

- ISO 8601 week format `"%Y-W%V"`:

```
std::cout << std::put_time(ltime, "%Y-W%V") << std::endl;
// 2016-W48
```

- ISO 8601 date with week number format `"%Y-W%V-%u"`:

```
std::cout << std::put_time(ltime, "%Y-W%V-%u") << std::endl;
// 2016-W48-7
```

- ISO 8601 ordinal date format `"%Y-%j"`:

```
std::cout << std::put_time(ltime, "%Y-%j") << std::endl;
// 2016-339
```

To read a time value from an input stream, you should perform the following steps:

1. Declare an object of the `std::tm` type to hold the time value read from the stream:

```
auto time = std::tm {};
```

2. Use `std::get_time()` supplying a pointer to the `std::tm` object that will hold the time value and a pointer to a null-terminated character string representing the format. The list of possible formats can be consulted at http://en.cppreference.com/w/cpp/io/manip/get_time. The following example parses an ISO 8601 combined date and time value:

```
std::istringstream stext("2016-12-04T05:26:47+0900");
stext >> std::get_time(&time, "%Y-%m-%dT%H:%M:%S");
if (!stext.fail()) { /* do something */ }
```

3. To read a standard date and time string according to the settings of a specific locale, first set the locale for the stream by calling `imblue()` and then use the `std::get_time()` manipulator:

```
std::istringstream stext("Sun 04 Dec 2016 05:35:30 JST");
stext.imbue(std::locale("en_GB.utf8"));
stext >> std::get_time(&time, "%c");
if (stext.fail()) { /* do something else */ }
```

How it works…

The two manipulators for time values, `put_time()` and `get_time()`, are very similar: they are both function templates with two arguments. The first argument is a pointer to an `std::tm` object representing the calendar date and time that holds the value to be written to the stream or the value read from the stream. The second argument is a pointer to a null-terminated character string representing the format of the time text. `put_time()` uses the `std::time_put()` facet to output a date and time value, and `get_time()` uses the `std::time_get()` facet to parse a date and time value. Both manipulator function templates return an object of an unspecified type. These functions do not throw exceptions:

```
template<class CharT>
/*unspecified*/ put_time(const std::tm* tmb, const CharT* fmt);

template<class CharT>
/*unspecified*/ get_time(std::tm* tmb, const CharT* fmt);
```

The string resulted from using `put_time()` to write a date and time value to an output stream is the same as resulted from a call to `std::strftime()` or `std::wcsftime()`.

The standard defines a long list of available conversion specifiers that compose the format string. These specifiers are prefixed with a `%`, in some cases followed by an `E` or a `0`. Some of them are also equivalent; for instance, `%F` is equivalent to `%Y-%m-%d` (that is the ISO 8601 date format) and `%T` is equivalent to `%H:%M:%S` (that is the ISO 8601 time format). The examples in this recipe mention only a few of the conversion specifiers, referring to ISO 8601 date and time formats. For the complete list of conversion specifiers, see the C++ standard or follow the links mentioned earlier.

It is important to note that not all conversion specifiers supported by `put_time()` are also supported by `get_time()`. Examples include `z` (offset from UTC in the ISO 8601 format) and `z` (time zone name or abbreviation) specifiers that can only be used with `put_time()`.

```
std::istringstream stext("2016-12-04T05:26:47+0900");
auto time = std::tm {};

stext >> std::get_time(&time, "%Y-%m-%dT%H:%M:%S%z"); // fails
stext >> std::get_time(&time, "%Y-%m-%dT%H:%M:%S");   // OK
```

The text represented by some conversion specifiers is locale dependent. All specifiers prefixed with `E` or `0` are locale dependent. To set a particular locale for the stream, use the `imbue()` method, as shown in the examples in the previous section.

See also

- *Using I/O manipulators to control the output of a stream*
- *Using monetary I/O manipulators*

Working with filesystem paths

An important addition to the C++17 standard is the `filesystem` library that enables us to work with paths, files, and directories in the hierarchical filesystems (such as Windows or POSIX filesystems). This standard library has been developed based on the `boost.filesystem` library. In the next few recipes, we will explore those features of the library that enable us to perform operations with files and directories, such as creating, moving, or deleting them, but also querying properties and searching. It is important, however, that we first look at how library handles paths.

Getting ready

For this recipe, we will consider most of the examples using Windows paths. In the accompanying code, all examples have both Windows and POSIX alternatives.

The `filesystem` library is available in the `std::filesystem` namespace in the `<filesystem>` header. To simplify code, we will use the following namespace alias in all examples:

```
namespace fs = std::filesystem;
```

At the time of writing this book all major compilers provide an implementation of the library, although it is considered to be still experimental and therefore provided in a namespace with this name. Because of that, the actual namespace for the library is `std::experimental::filesystem` and the actual header for GCC and Clang is `<experimental/filesystem>`.

A path to a filesystem component (file, directory, hard link, or soft link) is represented by the `path` class.

How to do it…

The following is a list of the most common operations on paths:

- Create a path using the constructor, assignment operator, or the `assign()` method:

```
// Windows
auto path = fs::path{"C:\\Users\\Marius\\Documents"};

// POSIX
auto path = fs::path{ "/home/marius/docs" };
```

- Append elements to a path by including a directory separator using member `operator /=`, non-member `operator /`, or the `append()` method:

```
path /= "Book";
path = path / "Modern" / "Cpp";
path.append("Programming");
// Windows: C:\Users\Marius\Documents\Book\Modern\Cpp\Programming
// POSIX:   /home/marius/docs/Book/Modern/Cpp/Programming
```

- Concatenate elements to a path without including a directory separator using member `operator +=`, non-member `operator +`, or the `concat()` method:

```
auto path = fs::path{ "C:\\Users\\Marius\\Documents" };
path += "Book";
path.concat("Modern");
// path = C:\Users\Marius\Documents\Book\Modern
```

- Decompose the elements of a path to its parts, such as root, root directory, parent path, filename, extension, and so on, using member functions such as `root_name()`, `root_dir()`, `filename()`, `stem()`, `extension()`, and so on (all of them are shown in the following example):

```
auto path =
  fs::path{"C:\\Users\\Marius\\Documents\\sample.file.txt"};

std::cout
  << "root: " << path.root_name() << std::endl
  << "root dir: " << path.root_directory() << std::endl
  << "root path: " << path.root_path() << std::endl
  << "rel path: " << path.relative_path() << std::endl
  << "parent path: " << path.parent_path() << std::endl
  << "filename: " << path.filename() << std::endl
  << "stem: " << path.stem() << std::endl
  << "extension: " << path.extension() << std::endl;
```

- Query if parts of a part are available using member functions such as `has_root_name()`, `has_root_directory()`, `has_filename()`, `has_stem()`, and `has_extension()` (all of them are shown in the following example):

```
auto path =
  fs::path{"C:\\Users\\Marius\\Documents\\sample.file.txt"};

std::cout
  << "has root: " << path.has_root_name() << std::endl
  << "has root dir: " << path.has_root_directory() << std::endl
  << "has root path: " << path.has_root_path() << std::endl
  << "has rel path: " << path.has_relative_path() << std::endl
```

```
  << "has parent path: " << path.has_parent_path() << std::endl
  << "has filename: " << path.has_filename() << std::endl
  << "has stem: " << path.has_stem() << std::endl
  << "has extension: " << path.has_extension() << std::endl;
```

- Check whether a path is relative or absolute:

```
auto path2 = fs::path{ "marius\\temp" };
std::cout
  << "absolute: " << path1.is_absolute() << std::endl
  << "absolute: " << path2.is_absolute() << std::endl;
```

- Modify individual parts of the path, such as filename with `replace_filename()` and `remove_filename()`, and extension with `replace_extension()`:

```
auto path =
  fs::path{"C:\\Users\\Marius\\Documents\\sample.file.txt"};

path.replace_filename("output");
path.replace_extension(".log");
// path = C:\Users\Marius\Documents\output.log

path.remove_filename();
// path = C:\Users\Marius\Documents
```

- Convert the directory separator to the system-preferred separator:

```
// Windows
auto path = fs::path{"Users/Marius/Documents"};
path.make_preferred();
// path = Users\Marius\Documents

// POSIX
auto path = fs::path{ "home/docs" };
path.make_preferred();
// path = /home/marius/docs
```

How it works…

The `std::filesystem::path` class models paths to filesystem components. However, it only handles the syntax and does not validate the existence of a component (such as file or directory) represented by the path.

The library defines a portable, generic syntax for paths that can accommodate various filesystems, such as POSIX or Windows, including the Microsoft Windows *UNC* (*Universal Naming Convention*) format. These two differ in several key aspects:

- POSIX systems have a single tree, no root name, a single root directory called `/`, and a single current directory and use `/` for directory separator. Paths are represented as null-terminated strings of `char` encoded as UTF-8.
- Windows systems have multiple trees, each with a root name (such as `c:`), a root directory (such as `c:`), and a current directory. Paths are represented as null-terminated strings of wide characters encoded as UTF-16.

A path name as defined in the `filesystem` library has the following syntax:

- An optional root name (`c:` or `//localhost`)
- An optional root directory
- Zero or more filenames (that may refer to a file, directory, hard link, or symbolic link) or directory-separators

There are two special filenames that are recognized: single dot (`.`) that represents the current directory and the double dot (`..`) that represents the parent directory. The directory separator can be repeated, in which case it is treated as a single separator (in other words, `/home////docs` is the same as `/home/marius/docs`). A path that has no redundant current directory name (`.`), no redundant parent directory name (`..`), and no redundant directory separators is said to be in a normal form.

The path operations presented in the previous section are only the most common operations with paths. However, the implementation defines additional querying and modifying methods, iterators, non-member comparison operators, and others. The following sample iterates through the parts of a path and prints them to the console:

```
auto path =
  fs::path{ "C:\\Users\\Marius\\Documents\\sample.file.txt" };

for (auto const & part : path)
{
  std::cout << part << std::endl;
}
```

The following is its result:

```
C:

Users
Marius
Documents
sample.file.txt
```

In this example, `sample.file.txt` is the filename. This is basically the part from the last directory separator to the end of the path. This is what member function `filename()` would be returning for the given path. The extension for this file is `.txt`, which is the string returned by the `extension()` member function. To retrieve the filename without extension, another member function called `stem()` is available. For this example, the string returned by this method is `sample.file`. For all these methods, but also the other decomposition methods, there is a corresponding querying method with the same name and prefix `has_`, such as `has_filename()`, `has_stem()`, and `has_extension()`. All these methods return a `bool` value to indicate whether the path has the corresponding part.

See also

- *Creating, copying, and deleting files and directories*
- *Checking the properties of an existing file or directory*

Creating, copying, and deleting files and directories

Operations with files, such as copying, moving, and deleting, or with directories, such as creating, renaming, and deleting, are all supported by the `filesystem` library. Files and directories are identified with a path (that can be absolute, canonical, or relative), a topic that was covered in the previous recipes. In this recipe, we will look at what are the standard functions for the above-mentioned operations and how they work.

Getting ready

Before going forward, you should read the *Working with filesystem paths* recipe. Introductory notes from that recipe also apply here. However, all examples in this recipe are platform independent.

For all the following examples, we will use the following variables, and assume the current path is `C:\Users\Marius\Documents` on Windows, and `/home/marius/docs` for a POSIX system. We will also assume the presence of the file called `sample.txt` in the `temp` subdirectory of the current path (such as `C:\Users\Marius\Documents\temp\sample.txt` or `/home/marius/docs/temp/sample.txt`):

```
    auto err = std::error_code{};
    auto basepath = fs::current_path();
    auto path = basepath / "temp";
    auto filepath = path / "sample.txt";
```

How to do it…

Use the following library functions to perform operations with directories:

- To create a new directory, use `create_directory()`. This method does nothing if the directory already exists, but does not create directories recursively:

```
auto success = fs::create_directory(path, err);
```

- To create new directories recursively, use `create_directories()`:

```
auto temp = path / "tmp1" / "tmp2" / "tmp3";
auto success = fs::create_directories(temp, err);
```

- To move an existing directory, use `rename()`:

```
auto temp = path / "tmp1" / "tmp2" / "tmp3";
auto newtemp = path / "tmp1" / "tmp3";

fs::rename(temp, newtemp, err);
if (err) std::cout << err.message() << std::endl;
```

- To rename an existing directory, also use `rename()`:

```
auto temp = path / "tmp1" / "tmp3";
auto newtemp = path / "tmp1" / "tmp4";

fs::rename(temp, newtemp, err);
if (err) std::cout << err.message() << std::endl;
```

- To copy an existing directory, use `copy()`. To copy recursively the entire content of a directory, use the `copy_options::recursive` flag:

```
fs::copy(basepath / "temp", basepath / "temp2",
         fs::copy_options::recursive, err);
if (err) std::cout << err.message() << std::endl;
```

- To create a symbolic link to a directory, use `create_directory_symlink()`:

```
auto linkdir = basepath / "templink";
fs::create_directory_symlink(path, linkdir, err);
if (err) std::cout << err.message() << std::endl;
```

- To remove an empty directory, use `remove()`:

```
auto temp = path / "tmp1" / "tmp4";
auto success = fs::remove(temp, err);
```

- To remove the entire content of a directory recursively and the directory itself, use `remove_all()`:

```
auto success = fs::remove_all(path, err) !=
               static_cast<std::uintmax_t>(-1);
```

Use the following library functions to perform operations with files:

- To copy a file, use `copy()` or `copy_file()`. The next section explains the difference between these two:

```
auto success = fs::copy_file(filepath, path / "sample.bak", err);
if (!success) std::cout << err.message() << std::endl;

fs::copy(filepath, path / "sample.cpy", err);
if (err) std::cout << err.message() << std::endl;
```

- To rename a file, use `rename()`:

```
auto newpath = path / "sample.log";
fs::rename(filepath, newpath, err);
if (err) std::cout << err.message() << std::endl;
```

- To move a file, use `rename()`:

```
auto newpath = path / "sample.log";
fs::rename(newpath, path / "tmp1" / "sample.log", err);
if (err) std::cout << err.message() << std::endl;
```

- To create a symbolic link to a file, use `create_symlink()`:

```
auto linkpath = path / "sample.txt.link";
fs::create_symlink(filepath, linkpath, err);
if (err) std::cout << err.message() << std::endl;
```

- To delete a file, use `remove()`:

```
auto success = fs::remove(path / "sample.cpy", err);
if (!success) std::cout << err.message() << std::endl;
```

How it works…

All the functions mentioned in this recipe, and other similar functions that are not discussed here, have multiple overloads grouped in two categories:

- Overloads that take as a last argument a reference to an `std::error_code`: these overloads do not throw an exception (they are defined with the `noexcept` specification), but instead set the value of the `error_code` object to the operating system error code, if an operating system error occurred. If no such error occurred, then the `clear()` method on the `error_code` object is called to reset any possible previously set code.
- Overloads that do not take the last argument of the `std::error_code` type: these overloads throw exceptions if errors occur. If an operating system error occurs, they throw an `std::filesystem::filesystem_error` exception. On the other hand, if memory allocation fails, these functions throw an `std::bad_alloc` exception.

All the examples in the previous section used the overload that does not throw exceptions, but instead set a code when an error occurs. Some functions return a `bool` to indicate success or failure. You can check whether the `error_code` object holds the code of an error either by checking whether the value of the error code, returned by method `value()`, is different than zero, or by using the conversion `operator bool`, that returns `true` for the same case, and `false` otherwise. To retrieve the explanatory string for the error code, use the `message()` method.

Some of the `filesystem` library functions are common for both files and directories. This is the case for `rename()`, `remove()`, and `copy()`. The working details of each of these functions can be complex, especially in the case of `copy()`, and are beyond the scope of this recipe. You should read the reference documentation if you need to perform anything other than the simple operations covered here.

When it comes to copying files, there are two functions that can be used: `copy()` and `copy_file()`. These have equivalent overloads with identical signatures and, apparently, work the same way. However, there is an important difference (other than the fact that `copy()` also works for directories): `copy_file()` follows symbolic links. To avoid doing that and copying the actual symbolic link, you must use either `copy_symlink()` or `copy()` with the `copy_options::copy_symlinks` flag. Both the `copy()` and `copy_file()` functions have an overload that takes an argument of the `std::filesystem::copy_options` type that defines how the operation should be performed. `copy_options` is a scoped `enum` with the following definition:

```
enum class copy_options
{
  none = 0,
  skip_existing = 1,
  overwrite_existing = 2,
  update_existing = 4,
  recursive = 8,
  copy_symlinks = 16,
  skip_symlinks = 32,
  directories_only = 64,
  create_symlinks = 128,
  create_hard_links = 256
```

```
    };
```

The following table defines how each of these flags affect a copy operation, either with `copy()` or `copy_file()`. The table is taken from the 27.10.10.4 paragraph of the C++17 standard:

Option group controlling copy_file function effects for existing target files	
`none`	(Default) Error; file already exists
`skip_existing`	Do not overwrite existing file; do not report an error
`overwrite_existing`	Overwrite the existing file
`update_existing`	Overwrite the existing file if it is older than the replacement file
Option group controlling copy function effects for subdirectories	
`none`	(Default) Do not copy subdirectories
`recursive`	Recursively copy subdirectories and their contents
Option group controlling copy function effects for symbolic links	
`none`	(Default) Follow symbolic links
`copy_symlinks`	Copy symbolic links as symbolic links rather than copying the files that they point to
`skip_symlinks`	Ignore symbolic links
Option group controlling copy function effects for choosing the form of copying	
`none`	(Default) Copy content
`directories_only`	Copy directory structure only, do not copy non-directory files
`create_symlinks`	Make symbolic links instead of copies of files; the source path shall be an absolute path unless the destination path is in the current directory
`create_hard_links`	Make hard links instead of copies of files

Another aspect that should be mentioned is related to symbolic links: `create_directory_symlink()` creates a symbolic link to a directory, whereas `create_symlink()`

creates symbolic links to either files or directories. On POSIX systems, the two are identical when it comes to directories. On other systems (such as Windows), symbolic links to directories are created differently than symbolic links to files. Therefore, it is recommended that you use `create_directory_symlink()` for directories, in order to write code that works correctly on all systems.

When you perform operations with files and directories such as the ones described in this recipe and you use the overloads that may throw exceptions, ensure that you try-catch the calls. Regardless of the type of overloads used, you should check the success of the operation and take appropriate action in case of failure.

See also

- *Working with filesystem paths*
- *Removing content from a file*
- *Checking the properties of an existing file or directory*

Removing content from a file

Operations such as copying, renaming, moving, or deleting files are directly provided by the `filesystem` library. However, when it comes to removing content from a file, you must perform explicit actions. Regardless of whether you need to do this for a text or binary files, you must implement the following pattern:

1. Create a temporary file.
2. Copy only the content that you want from the original file to the temporary file.
3. Delete the original file.
4. Rename/move the temporary file to the name/location of the original file.

Getting ready

In this recipe, we will see how to implement the pattern mentioned earlier for a text file. To do this, we will consider removing empty lines or lines that start with a semicolon (`;`). For this example, we will have an initial file called `sample.dat` that contains the names of Shakespeare's plays, but also empty lines and lines that start with a semicolon. The following is a partial listing of this file (from the beginning):

```
;Shakespeare's plays, listed by genre

;TRAGEDIES
Troilus and Cressida
Coriolanus
Titus Andronicus
Romeo and Juliet
Timon of Athens
Julius Caesar
```

The code samples listed in the next section use the following variables:

```
    auto path = fs::current_path();
    auto filepath = path / "sample.dat";
    auto temppath = path / "sample.tmp";
    auto err = std::error_code{};
```

How to do it…

Perform the following operations to remove content from a file:

1. Open the file for reading:

```
std::ifstream in(filepath);
if (!in.is_open())
{
  std::cout << "File could not be opened!" << std::endl;
  return;
}
```

2. Open another temporary file for writing; if the file already exists, truncate its content:

```
std::ofstream out(temppath, std::ios::trunc);
if (!out.is_open())
{
  std::cout << "Temporary file could not be created!"
            << std::endl;
  return;
}
```

3. Read line by line from the input file and copy the selected content to the output file:

```
auto line = std::string{};
while (std::getline(in, line))
{
  if (!line.empty() && line.at(0) != ';')
  {
    out << line << 'n';
  }
}
```

4. Close both input and output files:

```
in.close();
out.close();
```

5. Delete the original file:

```
auto success = fs::remove(filepath, err);
if(!success || err)
{
  std::cout << err.message() << std::endl;
  return;
}
```

6. Rename/move the temporary file to the name/location of the original file:

```
fs::rename(temppath, filepath, err);
if (err)
{
  std::cout << err.message() << std::endl;
}
```

How it works…

The pattern described above is the same for binary files too, but for our convenience, we are only discussing an example with text files. The temporary file in this example is in the same directory with the original file. Alternatively, this can be located in a separate directory, such as a user-temporary directory. To get a path to a temporary directory, you can use `std::filesystem::temp_directory_path()`. On Windows systems, this function returns the same directory as `GetTempPath()`. On POSIX systems, it returns the path specified in one of the environment variables `TMPDIR`, `TMP`, `TEMP`, or `TEMPDIR`, or if none of them are available, then it returns the path `/tmp`.

How content from the original file is copied to the temporary file varies from one case to another, depending on what needs to be copied. In the preceding example, we have copied entire lines, unless they are empty or start with a semicolon. For this purpose, we read the content of the original file line by line using `std::getline()` until there were no more lines to read. After all necessary content has been copied, the files should be closed, so they can be moved or deleted.

To complete the operation, there are three options:

- Delete the original file and rename the temporary file to the same name as the original one, if they were in the same directory, or move the temporary file to the original file location if they were in different directories. This is the approach taken in this recipe. For this, we used the `remove()` function to delete the original file and `rename()` to rename the temporary file to the original filename.
- Copy the content of the temporary file to the original file (for this, you can use either `copy()` or `copy_file()` functions) and then delete the temporary file (use `remove()` for that).
- Rename the original file (for instance, changing the extension or the name) and then use the original filename to rename/move the temporary file.

If you take the first approach mentioned above, then you must make sure that the temporary file that is later replacing the original file has the same file permissions as the original file, otherwise, depending on the context of your solution, it can lead to problems.

See also

- *Creating, copying, and deleting files and directories*

Checking the properties of an existing file or directory

The `filesystem` library provides functions and types that enable developers to check for the existence of a filesystem object, such as a file or directory, its properties, such as the type (file, directory, symbolic link, and so on), the last write time, permissions, and others. In this recipe, we will look at what these types and functions are and how they can be used.

Getting ready

Before continuing with this recipe, you should read the *Working with filesystem paths* recipe.

For the following code samples, we will use the namespace alias `fs` for the `std::filesystem` namespace. The `filesystem` library is available in the header with the same name, `<filesystem>`. Also, we will use the variables shown here, `path` for the path of a file and `err` for receiving potential operating system error codes from the filesystem APIs:

```
    auto path = fs::current_path() / "main.cpp";
    auto err = std::error_code{};
```

How to do it…

Use the following library functions to retrieve information about filesystem objects:

- To check whether a path refers to an existing filesystem object, use `exists()`:

```
auto exists = fs::exists(path, err);
std::cout << "file exists: " << std::boolalpha
          << exists << std::endl;
```

- To check whether two different paths refer to the same filesystem object, use `equivalent()`:

```
auto same = fs::equivalent(path,
               fs::current_path() / "." / "main.cpp");
std::cout << "equivalent: " << same << std::endl;
```

- To retrieve the size of a file in bytes, use `file_size()`:

```
auto size = fs::file_size(path, err);
std::cout << "file size: " << size << std::endl;
```

- To retrieve the count of hard links to a filesystem object, use `hard_link_count()`:

```
auto links = fs::hard_link_count(path, err);
if(links != static_cast<uintmax_t>(-1))
  std::cout << "hard links: " << links << std::endl;
else
  std::cout << "hard links: error" << std::endl;
```

- To retrieve or set the last modification time for a filesystem object, use `last_write_time()`:

```
auto lwt = fs::last_write_time(path, err);
auto time = decltype(lwt)::clock::to_time_t(lwt);
auto localtime = std::localtime(&time);
std::cout << "last write time: "
          << std::put_time(localtime, "%c") << std::endl;
```

- To retrieve POSIX file attributes, such as type and permissions, use the `status()` function. This function follows symbolic links. To retrieve the file attributes of a symbolic link without following it, use `symlink_status()`:

```
auto print_perm = [](fs::perms p)
{
  std::cout
    << ((p & fs::perms::owner_read) != fs::perms::none ?
       "r" : "-")
    << ((p & fs::perms::owner_write) != fs::perms::none ?
       "w" : "-")
    << ((p & fs::perms::owner_exec) != fs::perms::none ?
       "x" : "-")
    << ((p & fs::perms::group_read) != fs::perms::none ?
       "r" : "-")
    << ((p & fs::perms::group_write) != fs::perms::none ?
       "w" : "-")
    << ((p & fs::perms::group_exec) != fs::perms::none ?
       "x" : "-")
    << ((p & fs::perms::others_read) != fs::perms::none ?
       "r" : "-")
```

```
            << ((p & fs::perms::others_write) != fs::perms::none ?
               "w" : "-")
            << ((p & fs::perms::others_exec) != fs::perms::none ?
               "x" : "-")
            << std::endl;
        };

        auto status = fs::status(path, err);
        std::cout << "permissions: ";
        print_perm(status.permissions());
```

- To check whether a path refers to a particular type of filesystem object, such as file, directory, symbolic link, and so on, use functions `is_regular_file()`, `is_directory()`, `is_symlink()`, and so on:

```
        std::cout << "regular file? " <<
                  fs::is_regular_file(path, err) << std::endl;
        std::cout << "directory? " <<
                  fs::is_directory(path, err) << std::endl;
        std::cout << "char file? " <<
                  fs::is_character_file(path, err) << std::endl;
        std::cout << "symlink? " <<
                  fs::is_symlink(path, err) << std::endl;
```

How it works…

All the functions discussed in this recipe have an overload that throws exceptions upon error, and an overload that does not throw but returns an error code via a function parameter. All the examples in this recipe used this approach. More information about these sets of overloads can be found in the *Creating, copying, and deleting files and directories* recipe.

These functions used for retrieving information about the filesystem files and directories are in general simple and straightforward. However, some considerations are necessary:

- Checking whether a filesystem object exists can be done with `exists()` either by passing the path or an `std::filesystem::file_status` object that was previously retrieved using the `status()` function.
- The `equivalent()` function determines whether two filesystem objects have the same status, as retrieved by function `status()`. If neither paths exists, or if both exist but neither is a file, directory, or symbolic link, then the function returns an error. Hard links to the same file object are equivalent. A symbolic link and its target are also equivalent.
- The `file_size()` function can only be used to determine the size of regular files and symbolic links that target a regular file. For any other type of file objects, such as directories, this function fails. This function returns the size in bytes of the file, or -1 if an error occurred. If you want to determine whether a file is empty, you can use the `is_empty()` function. This works for all types of filesystem objects, including directories.
- The `last_write_time()` function has two sets of overloads: one that is used for retrieving the last modification time of the filesystem object, and one that is used to set the last modification time. Time is indicated by a `std::filesystem::file_time_type` object that is basically a type alias for `std::chrono::time_point`. The following example changes the last write time for a file to 30 minutes back than it used to be:

```
using namespace std::chrono_literals;
auto lwt = fs::last_write_time(path, err);
fs::last_write_time(path, lwt - 30min);
```

- The `status()` function determines the type and permissions of a filesystem object. However, if the file is a symbolic link, the information returned is about the target of the symbolic link. To retrieve information about the symbolic link itself, the `symlink_status()` function must be used. Permissions are defined as an enumeration, `std::filesystem::perms`. Not all the enumerators of this scoped `enum` represent permissions; some of them represent controlling bits, such as `add_perms` to indicate that permissions should be added, or `remove_perms` to indicate that permissions should be removed. The `permissions()` function can be used to modify permissions of a file or directory. The following example adds all permissions to the owner and user group of a file:

```
fs::permissions(
  path,
  fs::perms::add_perms |
  fs::perms::owner_all | fs::perms::group_all,
  err);
```

- To determine the type of a filesystem object, such as file, directory, or symbolic link, there are two options available: retrieve the file status and then check the type property, or use one of the available filesystem functions, such as `is_regular_file()`, `is_symlink()`, or `is_directory()`. The following examples that check whether a path refers to a regular file are equivalent:

```
auto s = fs::status(path, err);
auto isfile = s.type() == std::filesystem::file_type::regular;

auto isfile = fs::is_regular_file(path, err);
```

See also

- *Working with filesystem paths*
- *Creating, copying, and deleting files and directories*
- *Enumerating the content of a directory*

Enumerating the content of a directory

So far in this chapter, we took a look at many of the functionalities provided by the `filesystem` library, such as working with paths, performing operations with files and directories (creating, moving, renaming, deleting, and so on), and querying or modifying properties. Another useful functionality when working with the filesystem is to iterate through the content of a directory. The file library provides two directory iterators, one called `directory_iterator` that iterates the content of a directory, and one called `recursive_directory_iterator` that iterates recursively the content of a directory and its subdirectories. In this recipe, we will see how to use these.

Getting ready

In this recipe, we will work with filesystem paths and will check the properties of a filesystem object. Therefore, it is recommended that you first read the recipes *Working with filesystem paths* and *Checking the properties of an existing file or directory*.

For this recipe, we will consider a directory with the following structure:

```
test/
├──data/
│  ├──input.dat
│  └──output.dat
├──file_1.txt
├──file_2.txt
└──file_3.log
```

How to do it…

Use the following patterns to enumerate the content of a directory:

- To iterate only the content of a directory without recursively visiting its subdirectories, use `directory_iterator`:

```
void visit_directory(fs::path const & dir)
{
  if (fs::exists(dir) && fs::is_directory(dir))
  {
    for (auto const & entry : fs::directory_iterator(dir))
    {
      auto filename = entry.path().filename();
      if (fs::is_directory(entry.status()))
        std::cout << "[+]" << filename << std::endl;
      else if (fs::is_symlink(entry.status()))
        std::cout << "[>]" << filename << std::endl;
      else if (fs::is_regular_file(entry.status()))
        std::cout << " " << filename << std::endl;
      else
        std::cout << "[?]" << filename << std::endl;
     }
   }
 }
```

- To iterate all the content of a directory, including its subdirectories, use `recursive_directory_iterator` when the order of processing the entries does not matter:

```
void visit_directory_rec(fs::path const & dir)
{
  if (fs::exists(dir) && fs::is_directory(dir))
  {
    for (auto const & entry :
         fs::recursive_directory_iterator(dir))
    {
      auto filename = entry.path().filename();
      if (fs::is_directory(entry.status()))
        std::cout << "[+]" << filename << std::endl;
      else if (fs::is_symlink(entry.status()))
        std::cout << "[>]" << filename << std::endl;
      else if (fs::is_regular_file(entry.status()))
        std::cout << " " << filename << std::endl;
      else
        std::cout << "[?]" << filename << std::endl;
    }
  }
}
```

- To iterate all the content of a directory, including its subdirectories, in a structured manner, such as traversing a tree, use a function similar to the one in the first example, that uses `directory_iterator` to iterate the content of a directory, but call it recursively for each subdirectory:

```
void visit_directory(
  fs::path const & dir,
  bool const recursive = false,
  unsigned int const level = 0)
{
  if (fs::exists(dir) && fs::is_directory(dir))
  {
    auto lead = std::string(level*3, ' ');
    for (auto const & entry : fs::directory_iterator(dir))
    {
```

```
            auto filename = entry.path().filename();
            if (fs::is_directory(entry.status()))
            {
              std::cout << lead << "[+]" << filename << std::endl;
              if(recursive)
                visit_directory(entry, recursive, level+1);
            }
            else if (fs::is_symlink(entry.status()))
              std::cout << lead << "[>]" << filename << std::endl;
            else if (fs::is_regular_file(entry.status()))
              std::cout << lead << " " << filename << std::endl;
            else
              std::cout << lead << "[?]" << filename << std::endl;
          }
        }
      }
```

How it works…

Both `directory_iterator` and `recursive_directory_iterator` are input iterators that iterate over the entries of a directory. The difference is that the first one does not visit the subdirectories recursively, while the second one, as the name implies, does. The two have similar behavior:

- The order of iteration is unspecified.
- Each directory entry is visited only once.
- The special paths dot (`.`) and dot-dot (`..`) are skipped.
- A default constructed iterator is the end iterator and two end iterators are always equal.
- When iterated pass the last directory entries, it becomes equal to the end iterator.
- The standard does not specify what happens if a directory entry is added or deleted to the iterated directory after the iterator has been created.
- The standard defines non-member functions `begin()` and `end()` for both `directory_iterator` and `recursive_directory_iterator`, which enables us to use these iterators in range-based `for` loops, as seen in the examples earlier.

Both iterators have overloaded constructors. Some overloads of the `recursive_directory_iterator` constructor take an argument of the `std::filesystem::directory_options` type that specifies additional options for the iteration:

- `none`: This is the default that does not specify anything.
- `follow_directory_symlink`: This specifies that iteration should follow symbolic links instead of serving the link itself.
- `skip_permission_denied`: This specifies to ignore and skip the directories that would trigger an access denied error.

The elements both directory iterators point to are of the `directory_entry` type. The `path()` member function returns the path of the filesystem object represented by this object. The status of the filesystem object can be retrieved with member functions `status()` and `symlink_status()` for symbolic links.

The preceding examples follow a common pattern:

- Verify that the path to iterate actually exists.
- Use a range-based `for` loop to iterate all the entries of a directory.
- Use one of the two directory iterators available in the `filesystem` library, depending on the way iteration is supposed to be done.
- Process each entry according to the requirements.

In our examples, we simply printed the names of the directory entries to the console. It is important to note, as already specified earlier, that the content of the directory is iterated in an unspecified order. If you want to process the content in a structured manner, such

as showing subdirectories and their entries indented (for this particular case) or in a tree (in other types of applications), then using `recursive_directory_iterator` is not appropriate. Instead, you should use `directory_iterator` in a function that is called recursively from the iteration, for each subdirectory, as shown in the last example in the previous section.

Considering the directory structure presented at the beginning of this recipe (relative to the current path), we get the following output when using the recursive iterator as shown in the following example:

```
    visit_directory_rec(fs::current_path() / "test");
```

```
[+]data
   input.dat
   output.dat
   file_1.txt
   file_2.txt
   file_3.log
```

On the other hand, when using the recursive function from the third example as shown below, the output is displayed ordered on sublevels, as intended:

```
    visit_directory(fs::current_path() / "test", true);
```

```
[+]data
      input.dat
      output.dat
   file_1.txt
   file_2.txt
   file_3.log
```

There’s more…

In the previous recipe, *Checking the properties of an existing file or directory*, we have discussed, among others, about the `file_size()` function that returns the size in bytes of a file. However, this function fails if the specified path is a directory. To determine the size of a directory, we need to iterate recursively through the content of a directory, retrieve the size of regular files or symbolic links, and add them together. However, we must make sure that we check the value returned by `file_size()`, that is `-1` cast to an `std::uintmax_t`, in the case of an error. This value indicating failure, should not be added to the total size of a directory. The following function also returns `-1` as an `uintmax_t` in the case of an error:

```
std::uintmax_t dir_size(fs::path const & path)
{
  auto size = static_cast<uintmax_t>(-1);
  if (fs::exists(path) && fs::is_directory(path))
  {
    for (auto const & entry : fs::recursive_directory_iterator(path))
    {
      if (fs::is_regular_file(entry.status()) ||
      fs::is_symlink(entry.status()))
      {
        auto err = std::error_code{};
        auto filesize = fs::file_size(entry);
        if (filesize != static_cast<uintmax_t>(-1))
          size += filesize;
      }
    }
  }

  return size;
}
```

See also

- *Checking the properties of an existing file or directory*
- *Finding a file*

Finding a file

In the previous recipe, we saw how we can use `directory_iterator` and `recursive_directory_iterator` to enumerate the content of a directory. Displaying the content of a directory as we did in the previous recipe is only one of the scenarios where this is needed. The other major scenario is searching for particular entries in a directory, such as files with a particular name, extension, and so on. In this recipe, we will see how we can use the directory iterators and the iterating patterns shown earlier to find files that match a given criteria.

Getting ready

You should read the previous recipe, *Enumerating the content of a directory*, for details about directory iterators. In this recipe, we will also use the same test directory structure presented in the previous recipe.

How to do it…

To find files that match particular criteria, use the following pattern, exemplified in the `find_files()` function, as follows:

1. Use `recursive_directory_iterator` to iterate through all the entries of a directory and recursively through its subdirectories.
2. Consider regular files (and any other type of files you may need to process).
3. Use a function object (such as a lambda expression) to filter only the files that match your criteria.
4. Add the selected entries to a range (such as a vector):

```
std::vector<fs::path> find_files(
   fs::path const & dir,
   std::function<bool(fs::path const&)> filter)
{
  auto result = std::vector<fs::path>{};

  if (fs::exists(dir))
  {
    for (auto const & entry :
      fs::recursive_directory_iterator(
        dir,
        fs::directory_options::follow_directory_symlink))
    {
      if (fs::is_regular_file(entry) &&
         filter(entry))
      {
        result.push_back(entry);
      }
    }
  }

  return result;
}
```

How it works…

When we want to find files in a directory, the structure of the directory and the order its entries, including subdirectories, are visited in is probably not important. Therefore, we can use the `recursive_directory_iterator` to iterate through the entries.

The function , `find_files()`, takes two arguments: a path and a function wrapper that is used to select the entries that should be returned. The return type is a vector of `filesystem::path` though, alternatively, it could also be a vector of `filesystem::directory_entry`. The recursive directory iterator used in this example does not follow symbolic links, returning the link itself and not the target. This behavior can be changed, using a constructor overload that has an argument of type `filesystem::directory_options` and passing `follow_directory_symlink`.

In the preceding example, we consider only regular files and ignore the other type of filesystem objects. The predicate is applied to the directory entry and, if it returns `true`, the entry is added to the result.

The following example uses the `find_files()` function to find all the files in the test directory that start with the prefix `file_`:

```
    auto results = find_files(
              fs::current_path() / "test",
              [](fs::path const & p) {
      auto filename = p.wstring();
      return filename.find(L"file_") != std::wstring::npos;
    });

    for (auto const & path : results)
    {
      std::cout << path << std::endl;
    }
```

The output of executing this program, with paths relative to the current path, is as follows:

```
test\file_1.txt
test\file_2.txt
test\file_3.log
```

A second example shows how to find files that have a particular extension, in this case, extension `.dat`:

```
    auto results = find_files(
           fs::current_path() / "test",
           [](fs::path const & p) {
      return p.extension() == L".dat";
    });

    for (auto const & path : results)
    {
      std::cout << path << std::endl;
    }
```

The output, again relative to the current path, is shown here:

```
test\data\input.dat
test\data\output.dat
```

See also

- *Checking the properties of an existing file or directory*
- *Enumerating the content of a directory*

Leveraging Threading and Concurrency

This chapter includes the following recipes:

- Working with threads
- Handling exceptions from thread functions
- Synchronizing access to shared data with mutexes and locks
- Avoiding using recursive mutexes
- Sending notifications between threads
- Using promises and futures to return values from threads
- Executing functions asynchronously
- Using atomic types
- Implementing parallel map and fold with threads
- Implementing parallel map and fold with tasks

Introduction

All computers contain multiple processors or at least multiple cores, and leveraging this computational power is the key for many categories of applications. Unfortunately, many developers still have a mindset of sequential code execution, even though operations that do not depend on each other could be executed concurrently. This chapter presents standard library support for threads, asynchronous tasks and related components, and some practical examples at the end.

Working with threads

Most modern processors (except those dedicated to types of applications that do not require great computing power, such as Internet of Things applications) have two, four, or more cores that enable you to concurrently execute multiple threads of execution. Applications must be explicitly written to leverage the multiple processing units that exist; you can write such applications by executing functions on multiple threads at the same time. The C++ standard library provides support for working with threads, synchronization of shared data, thread communication, and asynchronous tasks. This chapter explores the most important topics related to threads and tasks.

A thread is a sequence of instructions that can be managed independently by a scheduler, such as the operating system. Threads could be software; they can run on single processing units, usually by time slicing. They could be hardware as well; they can run simultaneously, that is, in parallel, on systems with multiprocessors or multicores. Many software threads can run concurrently on a hardware thread too. The C++ library provides support for working with software threads. In the first part of this chapter, we will look at the various threading objects and mechanisms that have built-in support in the library, such as threads, locking objects, condition variables, exception handling, and others. In this recipe, you will learn how to create and manage threads.

Getting ready

A thread of execution is represented by the `thread` class available in the `std` namespace in the `<thread>` header. Additional thread utilities are available in the same header but in the `std::this_thread` namespace.

In the following examples, the `print_time()` function is used:

```
inline void print_time()
{
  auto now = std::chrono::system_clock::now();
  auto stime = std::chrono::system_clock::to_time_t(now);
  auto ltime = std::localtime(&stime);

  std::cout << std::put_time(ltime, "%c") << std::endl;
}
```

How to do it…

Use the following solutions to manage threads:

- To create an `std::thread` object without starting the execution of a new thread, use its default constructor:

```
std::thread t;
```

- Start the execution of a function on another thread by constructing an `std::thread` object and passing the function as an argument:

```
void func1()
{
  std::cout << "thread func without params" << std::endl;
}

std::thread t(func1);
std::thread t([]() {
  std::cout << "thread func without params"
            << std::endl; });
```

- Start the execution of a function with arguments on another thread by constructing an `std::thread` object and passing the function as an argument to the constructor, followed by its arguments:

```
void func2(int const i, double const d, std::string const s)
{
  std::cout << i << ", " << d << ", " << s << std::endl;
}

std::thread t(func2, 42, 42.0, "42");
```

- To wait for a thread to finish the execution, use the `join()` method on the `thread` object:

```
t.join();
```

- To allow a thread to continue its execution independently of the current thread object, use the `detach()` method:

```
t.detach();
```

- To pass arguments by reference to a function, thread wrap them in either `std::ref` or `std::cref` (if the reference is constant):

```
void func3(int & i)
{
  i *= 2;
}

int n = 42;

std::thread t(func3, std::ref(n));
t.join();
std::cout << n << std::endl; // 84
```

- To stop the execution of a thread for a specified duration, use the

`std::this_thread::sleep_for()` function:

```
void func4()
{
  using namespace std::chrono;
  print_time();
  std::this_thread::sleep_for(2s);
  print_time();
}

std::thread t(func4);
t.join();
```

- To stop the execution of a thread until a specified moment in time, use the `std::this_thread::sleep_until()` function:

```
void func5()
{
  using namespace std::chrono;
  print_time();
  std::this_thread::sleep_until(
  std::chrono::system_clock::now() + 2s);
  print_time();
}

std::thread t(func5);
t.join();
```

- To suspend the execution of the current thread and provide an opportunity to another thread to perform the execution, use `std::this_thread::yield()`:

```
void func6(std::chrono::seconds timeout)
{
  auto now = std::chrono::system_clock::now();
  auto then = now + timeout;
  do
  {
    std::this_thread::yield();
  } while (std::chrono::system_clock::now() < then);
}

std::thread t(func6, std::chrono::seconds(2));
t.join();
print_time();
```

How it works…

The `std::thread` class that represents a single thread of execution has several constructors:

- A default constructor that only creates the thread object, but does not start the execution of a new thread.
- A move constructor that creates a new thread object to represent a thread of execution previously represented by the object it was constructed from. After the construction of the new object, the other object is no longer associated with the execution thread.
- A constructor with a variable number of arguments: the first being a function that represents the top-level thread function and the others being arguments to be passed to the thread function. Arguments need to be passed to the thread function by value. If the thread function takes parameters by reference or by constant reference, they must be wrapped in either an `std::ref` or `std::cref` object.

The thread function, in this case, cannot return a value. It is not illegal for the function to actually have a return type other than `void`, but it ignores any value that is directly returned by the function. If it has to return a value, it can do so using a shared variable or a function argument. In a future recipe, we will see how a thread function returns a value to another thread using a *promise*.

If the function terminates with an exception, the exception cannot be caught with a `try...catch` statement in the context where a thread was started and the program terminated abnormally with a call to `std::terminate()`. All exceptions must be caught within the executing thread, but they can be transported across threads via an `std::exception_ptr` object. We'll discuss this topic in the next recipe.

After a thread has started its execution, it is both joinable and detachable. Joining a thread implies blocking the execution of the current thread until the joined thread ends its execution. Detaching a thread means decoupling the thread object from the thread of execution it represents, allowing both the current thread and the detached thread to be executed at the same time. Joining a thread is done with `join()` and detaching a thread is done with `detach()`. Once you call either of these two methods, the thread is said to be non-joinable and the thread object can be safely destroyed. When a thread is detached, the shared data it may need to access must be available throughout its execution.
The `joinable()` method indicates whether a thread can be joined or not.

Each thread has an identifier that can be retrieved—for the current thread, call the `std::this_thread::get_id()` function; for another thread of execution represented by a `thread` object, call its `get_id()` method.

There are several additional utility functions available in the `std::this_thread` namespace:

- The `yield()` method hints the scheduler to activate another thread. This is useful when implementing a busy-waiting routine, as in the last example from the previous section.

- The `sleep_for()` method blocks the execution of the current thread for at least the specified period of time (the actual time the thread is put to sleep may be longer than the requested period due to scheduling).
- The `sleep_until()` method blocks the execution of the current thread until at least the specified time point (the actual duration of the sleep may be longer than requested due to scheduling).

See also

- *Handling exceptions from thread functions*
- *Synchronizing access to shared data with mutexes and locks*
- *Avoiding using recursive mutexes*
- *Sending notifications between threads*
- *Using promises and futures to return values from threads*

Handling exceptions from thread functions

In the previous recipe, we introduced the thread support library and saw how to do some basic operations with threads. In that recipe, we briefly discussed exception handling in thread functions and mentioned that exceptions cannot leave the top-level thread function because they cause the program to abnormally terminate with a call to `std::terminate()`. On the other hand, exceptions can be transported between threads within an `std::exception_ptr` wrapper; in this recipe, we will see how to do this.

Getting ready

You are now familiar with the thread operations we discussed in the previous recipe, *Working with threads*. The `exception_ptr` class is available in the `std` namespace, which is in the `<exception>` header; `mutex` (which we will discuss in more detail in the next recipe) is also available in the same namespace but in the `<mutex>` header.

How to do it…

To properly handle exceptions thrown in a worker thread from the main thread or the thread where it was joined, do the following (assuming multiple exceptions can be thrown from multiple threads):

1. Use a global container to hold instances of `std::exception_ptr`:

```
std::vector<std::exception_ptr> g_exceptions;
```

2. Use a global `mutex` to synchronize access to the shared container:

```
std::mutex g_mutex;
```

3. Use a `try...catch` block for the code that is being executed in the top-level thread function. Use `std::current_exception()` to capture the current exception and wrap a copy or its reference into an `std::exception_ptr` pointer, which is added to the shared container for exceptions:

```
void func1()
{
  throw std::exception("exception 1");
}

void func2()
{
  throw std::exception("exception 2");
}

void thread_func1()
{
  try
  {
    func1();
  }
  catch (...)
  {
    std::lock_guard<std::mutex> lock(g_mutex);
    g_exceptions.push_back(std::current_exception());
  }
}

void thread_func2()
{
  try
  {
    func2();
  }
  catch (...)
  {
    std::lock_guard<std::mutex> lock(g_mutex);
    g_exceptions.push_back(std::current_exception());
  }
}
```

4. Clear the container from the main thread before you start the threads:

```
g_exceptions.clear();
```

5. In the main thread, after the execution of all the threads has finished, inspect the

caught exceptions and handle each of them appropriately:

```
std::thread t1(thread_func1);
std::thread t2(thread_func2);
t1.join();
t2.join();

for (auto const & e : g_exceptions)
{
  try
  {
    if(e != nullptr)
      std::rethrow_exception(e);
  }
  catch(std::exception const & ex)
  {
    std::cout << ex.what() << std::endl;
  }
}
```

How it works…

For the example in the preceding section, we assumed that multiple threads can throw exceptions and therefore need a container to hold them all. If there is a single exception from a single thread at a time, then you do not need a shared container and a `mutex` to synchronize access to it. You can use a single global object of the type `std::exception_ptr` to hold the exception transported between threads.

The `std::current_exception()` is a function that is typically used in a `catch` clause to capture the current exception and create an instance of `std::exception_ptr`. This is done to hold a copy or reference (depending on the implementation) to the original exception, which remains valid as long as there is an `std::exception_ptr` pointer available that refers to it. If this function is called when no exception is being handled, then it creates an empty `std::exception_ptr`.

The `std::exception_ptr` pointer is a wrapper for an exception captured with `std::current_exception()`. If default constructed, it does not hold any exception. Two objects of this type are equal if they are both empty or point to the same exception object. The `std::exception_ptr` objects can be passed to other threads where they can be rethrown and caught in a `try...catch` block.

The `std::rethrow_exception()` is a function that takes `std::exception_ptr` as an argument and throws the exception object referred to by its argument.

`std::current_exception()`, `std::rethrow_exception()`, and `std::exception_ptr` are all available in C++11.

In the example from the previous section, each thread function uses a `try...catch` statement for the entire code it executes so that no exception may leave the function uncaught. When an exception is handled, a lock on the global `mutex` object is acquired and the `std::exception_ptr` object holding the current exception is added to the shared container. With this approach, the thread function stops at the first exception; however, in other circumstances, you may need to execute multiple operations even if the previous one throws an exception. In this case, you will have multiple `try...catch` statements and perhaps transport only some of the exceptions outside the thread. In the main thread, after all the threads have finished executing, the container is iterated and each non-empty exception is rethrown and caught with a `try...catch` block and handled appropriately.

See also

- *Working with threads*

Synchronizing access to shared data with mutexes and locks

Threads allow you to execute multiple functions at the same time, but it is often necessary that these functions access shared resources. Access to shared resources must be synchronized so that at a time, only one thread would be able to read or write from or to the shared resource. An example of this was shown in the previous recipe, where multiple threads had the ability to add objects to a shared container at the same time. In this recipe, we will see what are the mechanisms the C++ standard defines for synchronizing thread access with shared data and how they work.

Getting ready

The `mutex` and `lock` classes discussed in this recipe are available in the `std` namespace in the `<mutex>` header.

How to do it…

Use the following pattern for synchronizing access with a single shared resource:

1. Define a `mutex` in the appropriate context (class or global scope):

```
std::mutex g_mutex;
```

2. Acquire a `lock` on the `mutex` before accessing the shared resource in each thread:

```
void thread_func()
{
  using namespace std::chrono_literals;
  {
    std::lock_guard<std::mutex> lock(g_mutex);
    std::cout << "running thread "
              << std::this_thread::get_id() << std::endl;
  }

  std::this_thread::yield();
  std::this_thread::sleep_for(2s);

  {
    std::lock_guard<std::mutex> lock(g_mutex);
    std::cout << "done in thread "
              << std::this_thread::get_id() << std::endl;
  }
}
```

Use the following pattern for synchronizing access to multiple shared resources at the same time to avoid deadlocks:

1. Define a mutex for each shared resource in the appropriate context (global or class scope):

```
template <typename T>
struct container
{
  std::mutex     mutex;
  std::vector<T> data;
};
```

2. Lock the mutexes at the same time using a deadlock avoidance algorithm with `std::lock()`:

```
template <typename T>
void move_between(container<T> & c1, container<T> & c2,
                  T const value)
{
  std::lock(c1.mutex, c2.mutex);
  // continued at 3.
}
```

3. After locking them, adopt the ownership of each mutex into an `std::lock_guard` class to ensure they are safely released at the end of the function (or scope):

```
// continued from 2.
std::lock_guard<std::mutex> l1(c1.mutex, std::adopt_lock);
std::lock_guard<std::mutex> l2(c2.mutex, std::adopt_lock);

c1.data.erase(
```

```
          std::remove(c1.data.begin(), c1.data.end(), value),
          c1.data.end());
        c2.data.push_back(value);
```

How it works…

A mutex is a synchronization primitive that allows us to protect simultaneous access to shared resources from multiple threads. The C++ standard library provides several implementations:

- `std::mutex` is the most commonly used mutex type; it is illustrated in the preceding code snippet. It provides methods to acquire and release the mutex. `lock()` tries to acquire the mutex and blocks it if it is not available, `try_lock()` tries to acquire the mutex and returns it without blocking if the mutex is not available, and `unlock()` releases the mutex.
- `std::timed_mutex` is similar to `std::mutex` but provides two more methods to acquire the mutex using a timeout: `try_lock_for()` tries to acquire the mutex and returns it if the mutex is not made available during the specified duration, and `try_lock_until()` tries to acquire the mutex and returns it if the mutex is not made available until a specified time point.
- `std::recursive_mutex` is similar to `std::mutex`, but the mutex can be acquired multiple times from the same thread without being blocked.
- `std::recursive_timed_mutex` is a combination of a recursive mutex and a timed mutex.

The first thread that locks an available mutex takes ownership of it and continues with the execution. All consecutive attempts to lock the mutex from any thread fail, including the thread that already owns the mutex, and the `lock()` method blocks the thread until the mutex is released with a call to `unlock()`. If a thread needs to be able to lock a mutex multiple times without blocking it and therefore enter a deadlock, a `recursive_mutex` class template should be used.

The typical use of a mutex to protect access to a shared resource comprises locking the mutex, using the shared resource, and then unlocking the mutex:

```
g_mutex.lock();

// use the shared resource such as std::cout
std::cout << "accessing shared resource" << std::endl;

g_mutex.unlock();
```

This method of using the mutex is, however, prone to error. This is because each call to `lock()` must be paired with a call to `unlock()` on all execution paths, that is both normal return paths and exception return paths. In order to safely acquire and release a mutex, regardless of the way the execution of a function goes, the C++ standard defines several locking classes:

- `std::lock_guard` is the locking mechanism seen earlier; it represents a mutex wrapper implemented in an RAII manner. It attempts to acquire the mutex at the time of its construction and release it upon destruction. This is available in C++11. The following is a typical implementation of `lock_guard`:

```
template <class M>
class lock_guard
{
public:
  typedef M mutex_type;

  explicit lock_guard(M& Mtx) : mtx(Mtx)
  {
    mtx.lock();
  }

  lock_guard(M& Mtx, std::adopt_lock_t) : mtx(Mtx)
  { }

  ~lock_guard() noexcept
  {
    mtx.unlock();
  }

  lock_guard(const lock_guard&) = delete;
  lock_guard& operator=(const lock_guard&) = delete;
private:
  M& mtx;
};
```

- `std::unique_lock` is a mutex ownership wrapper that provides support for deferred locking, time locking, recursive locking, transfer of ownership, and using it with condition variables. This is available in C++11.
- `std::shared_lock` is a mutex-shared ownership wrapper that provides support for deferred locking, time locking, and transfer of ownership. This is available in C++14.
- `std::scoped_lock` is a wrapper for multiple mutexes implemented in an RAII manner: upon construction, it attempts to acquire ownership of the mutexes in a deadlock avoidance manner as if it is using `std::lock()`, and upon destruction, it releases the mutexes in reverse order of the way they were acquired. This is available in C++17.

In the first example of the *How to do it…* section, we used `std::mutex` and `std::lock_guard` to protect access to the `std::cout` stream object, which is shared between all the threads in a program. The following example shows how the `thread_func()` function can be executed concurrently on several threads:

```
std::vector<std::thread> threads;
for (int i = 0; i < 5; ++i)
  threads.emplace_back(thread_func);

for (auto & t : threads)
  t.join();
```

A possible output of the program is as follows:

```
running thread 140296854550272
running thread 140296846157568
running thread 140296837764864
running thread 140296829372160
running thread 140296820979456
done in thread 140296854550272
done in thread 140296846157568
done in thread 140296837764864
done in thread 140296820979456
done in thread 140296829372160
```

When a thread needs to take ownership of multiple mutexes that are meant for protecting multiple shared resources, acquiring them one by one may lead to deadlocks. Let’s consider the following example (where `container` is the class shown in the *How to do it…*

section):

```
template <typename T>
void move_between(container<T> & c1, container<T> & c2, T const value)
{
  std::lock_guard<std::mutex> l1(c1.mutex);
  std::lock_guard<std::mutex> l2(c2.mutex);

  c1.data.erase(
    std::remove(c1.data.begin(), c1.data.end(), value),
    c1.data.end());
  c2.data.push_back(value);
}

container<int> c1;
c1.data.push_back(1);
c1.data.push_back(2);
c1.data.push_back(3);

container<int> c2;
c2.data.push_back(4);
c2.data.push_back(5);
c2.data.push_back(6);

std::thread t1(move_between<int>, std::ref(c1), std::ref(c2), 3);
std::thread t2(move_between<int>, std::ref(c2), std::ref(c1), 6);

t1.join();
t2.join();
```

In this example, the `container` class holds data that may be accessed simultaneously from different threads; therefore, it needs to be protected by acquiring a mutex. The `move_between()` function is a thread-safe function that removes an element from a container and adds it to a second container. To do so, it acquires the mutexes of the two containers sequentially, then erases the element from the first container and adds it to the end of the second container.

This function is, however, prone to deadlocks because a race condition might be triggered while acquiring the locks. Suppose we have a scenario where two different threads execute this function, but with different arguments:

- The first thread starts executing with the arguments `c1` and `c2` in this order.
- The first thread is suspended after it acquires the lock for the `c1` container. The second thread starts executing with the arguments `c2` and `c1` in this order.
- The second thread is suspended after it acquires the lock for the `c2` container.
- The first thread continues the execution and tries to acquire the mutex for `c2`, but the mutex is unavailable. Therefore, a deadlock occurs (this can be simulated by putting the thread to sleep for a short while after it acquires the first mutex).

To avoid possible deadlocks such as these, mutexes should be acquired in a deadlock avoidance manner, and the standard library provides a utility function called `std::lock()` that does that. The `move_between()` function needs to change by replacing the two locks with the following code (as shown in the *How to do it…* section):

```
std::lock(c1.mutex, c2.mutex);

std::lock_guard<std::mutex> l1(c1.mutex, std::adopt_lock);
std::lock_guard<std::mutex> l2(c2.mutex, std::adopt_lock);
```

The ownership of the mutexes must still be transferred to a lock guard object so they are

properly released after the execution of the function ends (or depending on the case, when a particular scope ends).

In C++17, a new mutex wrapper is available, `std::scoped_lock`, that can be used to simplify code, such as the one in the preceding example. This type of lock can acquire the ownership of multiple mutexes in a deadlock-free manner. These mutexes are released when the scoped lock is destroyed. The preceding code is equivalent to the following single line of code:

```
    std::scoped_lock lock(c1.mutex, c2.mutex);
```

See also

- *Working with threads*
- *Avoiding using recursive mutexes*

Avoiding using recursive mutexes

The standard library provides several mutex types for protecting access to shared resources. `std::recursive_mutex` and `std::recursive_timed_mutex` are two implementations that allow you to use multiple locking in the same thread. A typical use of a recursive mutex is to protect access to a shared resource from a recursive function. Recursive mutexes have a greater overhead than non-recursive mutexes and, when possible, they should be avoided. This recipe presents a use case for transforming a thread-safe type using a recursive mutex into a thread-safe type using a non-recursive mutex.

Getting ready

You need to be familiar with the various mutexes and locks available in the standard library. I recommend that you read the previous recipe, *Synchronizing access to shared data with mutex and locks*, to get an overview of them.

The purpose of this recipe is to transform the following class so we can avoid using `std::recursive_mutex`:

```
class foo_rec
{
  std::recursive_mutex m;
  int data;

public:
  foo_rec(int const d = 0) : data(d) {}

  void update(int const d)
  {
    std::lock_guard<std::recursive_mutex> lock(m);
    data = d;
  }

  int update_with_return(int const d)
  {
    std::lock_guard<std::recursive_mutex> lock(m);
    auto temp = data;
    update(d);
    return temp;
  }
};
```

How to do it…

To transform the preceding implementation into a thread-safe type using a non-recursive mutex, do this:

1. Replace `std::recursive_mutex` with `std::mutex`:

```
class foo
{
  std::mutex m;
  int        data;
  // continued at 2.
};
```

2. Define private non-thread-safe versions of the public methods or helper functions to be used in thread-safe public methods:

```
void internal_update(int const d) { data = d; }
// continued at 3.
```

3. Rewrite the public methods to use the newly defined non-thread-safe private methods:

```
public:
  foo(int const d = 0) : data(d) {}
  void update(int const d)
  {
    std::lock_guard<std::mutex> lock(m);
    internal_update(d);
  }

  int update_with_return(int const d)
  {
    std::lock_guard<std::mutex> lock(m);
    auto temp = data;
    internal_update(d);
    return temp;
  }
```

How it works…

An `std::recursive_mutex` class may be locked multiple times from a thread, either with a call to `lock()` or `try_lock()`. When a thread locks an available recursive mutex, it acquires its ownership; as a result of this, consecutive attempts to lock the mutex from the same thread do not block the execution of the thread, creating a deadlock. The recursive mutex is, however, released only when an equal number of calls to `unlock()` are made.

The `foo_rec` class we just discussed uses a recursive mutex to protect access to shared data; in this case, it is an integer member variable that is accessed from two thread-safe public functions:

- `update()` sets a new value in the private variable.
- `update_and_return()` sets a new value in the private variable and returns the previous value to the called function. This function calls `update()` to set the new value.

The implementation of `foo_rec` was probably intended to avoid duplication of code, yet this particular approach is rather a design error that can be improved, as shown in the *How to do it…* section. Rather than reusing public thread-safe functions, we can provide private non-thread-safe functions that could then be called from the public interface.

The same solution can be applied to other similar problems: define a non-thread-safe version of the code and then provide, perhaps lightweight, thread-safe wrappers.

See also

- *Working with threads*
- *Synchronizing access to shared data with mutexes and locks*

Sending notifications between threads

Mutexes are synchronization primitives that can be used to protect access to shared data. However, the standard library provides a synchronization primitive called a *condition variable* that enables a thread to signal to others that a certain condition has occurred. The thread or the threads that are waiting on the condition variable are blocked until the condition variable is signaled or until a timeout or a spurious wakeup occurs. In this recipe, we will see how to use condition variables to send notifications between thread-producing data and thread-consuming data.

Getting ready

For this recipe, you need to be familiar with threads, mutexes, and locks. Condition variables are available in the `std` namespace in the `<condition_variable>` header.

How to do it…

Use the following pattern for synchronizing threads with notifications on condition variables:

1. Define a condition variable (in the appropriate context):

```
std::condition_variable cv;
```

2. Define a mutex for threads to lock on:

```
std::mutex cv_mutex;
```

3. Define the shared data used between the threads:

```
int data = 0;
```

4. In the producing thread, lock the mutex before you modify the data:

```
std::thread p([&](){
  // simulate long running operation
  {
    using namespace std::chrono_literals;
    std::this_thread::sleep_for(2s);
  }

  // produce
  {
    std::unique_lock lock(cv_mutex);
    data = 42;
  }

  // print message
  {
    std::lock_guard l(io_mutex);
    std::cout << "produced " << data << std::endl;
  }

  // continued at 5.
});
```

5. In the producing thread, signal the condition variable with a call to `notify_one()` or `notify_all()` (do this after the mutex used to protect the shared data is unlocked):

```
// continued from 4.
cv.notify_one();
```

6. In the consuming thread, acquire a unique lock on the mutex and use it to wait on the condition variable:

```
std::thread c([&](){
  // wait for notification
  {
    std::unique_lock lock(cv_mutex);
    cv.wait(lock);
  }

  // continued at 7.
});
```

7. In the consuming thread, use the shared data after the condition is notified:

```
        // continued from 6.
        {
          std::lock_guard lock(io_mutex);
          std::cout << "consumed " << data << std::endl;
        }
```

How it works…

The preceding example represents two threads that share common data (in this case, an integer variable). One thread produces data after a lengthy computation (simulated with a sleep), and the other consumes it only after it is produced. To do so, they use a synchronization mechanism that uses a mutex and a condition variable that blocks the consuming thread until a notification arises from the producer thread indicating that data has been made available. The key in this communication channel is the condition variable that the consuming thread waits on until the producing thread notifies it. Both threads start about the same time. The producer thread begins a long computation that is supposed to produce data for the consuming thread. At the same time, the consuming thread cannot actually proceed until the data is made available; it must remain blocked until it is notified that the data has been produced. Once notified, it can continue its execution. The entire mechanism works as follows:

- There must be at least one thread waiting on the condition variable to be notified.
- There must be at least one thread that is signaling the condition variable.
- The waiting threads must first acquire a lock on a mutex (`std::unique_lock<std::mutex>`) and pass it to the `wait()`, `wait_for()`, or `wait_until()` method of the condition variable. All the waiting methods atomically release the mutex and block the thread until the condition variable is signaled. At this point, the thread is unblocked and the mutex is atomically acquired again.
- The thread that signals the condition variable can do so with either `notify_one()`, where one blocked thread is unblocked, or `notify_all()`, where all the blocked threads waiting for the condition variable are unblocked.

Condition variables cannot be made completely predictable on multiprocessor systems. Therefore, spurious wakeups may occur and a thread is unlocked even if nobody signals the condition variable. So, it is necessary to check whether the condition is true after the thread has been unblocked. However, spurious wakeups may occur multiple times and, therefore, it is necessary to check the condition variable in a loop.

The C++ standard provides two implementations of condition variables:

- `std::condition_variable`, used in this recipe, defines a condition variable associated with `std::unique_lock`.
- `std::condition_variable_any` represents a more general implementation that works with any lock that meets the requirements of a basic lock (implements `lock()` and `unlock()` methods). A possible use of this implementation is providing interruptible waits, as explained by Anthony Williams in *C++ concurrency in action* (2012):

> *A custom lock operation would both lock the associated mutex as expected and also perform the necessary job of notifying this condition variable when the*

interrupting signal is received.

All the waiting methods of the condition variable have two overloads:

- The first overload takes `std::unique_lock<std::mutex>` (based on the type, that is, duration or time point) and causes the thread to remain blocked until the condition variable is signaled. This overload atomically releases the mutex and blocks the current thread and adds it to the list of threads waiting on the condition variable. The thread is unblocked when the condition is notified with either `notify_one()` or `notify_all()`, a spurious wakeup occurs, or a timeout occurs (depending on the function overload). When this happens, the mutex is atomically acquired again.
- The second overload takes a predicate in addition to the arguments of the other overloads. This predicate can be used to avoid spurious wakeups while waiting for a condition to become `true`. This overload is equivalent to the following:

```
      while(!pred())
        wait(lock);
```

The following code illustrates a similar but more complex example than the one presented in the previous section. The producing thread generates data in a loop (in this example, it is a finite loop), and the consuming thread waits for new data to be made available and consumes it (prints it to the console). The producing thread terminates when it finishes producing data, and the consuming thread terminates when there is no more data to consume. Data is added to `queue<int>`, and a Boolean variable is used to indicate to the consuming thread that the process of producing data is finished:

```
std::mutex g_lockprint;
std::mutex g_lockqueue;
std::condition_variable g_queuecheck;
std::queue<int> g_buffer;
bool g_done;

void producer(
 int const id,
 std::mt19937& generator,
 std::uniform_int_distribution<int>& dsleep,
 std::uniform_int_distribution<int>& dcode)
{
  for (int i = 0; i < 5; ++i)
  {
    // simulate work
    std::this_thread::sleep_for(
      std::chrono::milliseconds(dsleep(generator)));

    // generate data
    {
      std::unique_lock<std::mutex> locker(g_lockqueue);
      int value = id * 100 + dcode(generator);
      g_buffer.push(value);

      {
        std::unique_lock<std::mutex> locker(g_lockprint);
        std::cout << "[produced(" << id << ")]: " << value
                  << std::endl;
      }
    }

    // notify consumers
    g_queuecheck.notify_one();
  }
}
```

```
void consumer()
{
  // loop until end is signaled
  while (!g_done)
  {
    std::unique_lock<std::mutex> locker(g_lockqueue);

    g_queuecheck.wait_for(
      locker,
      std::chrono::seconds(1),
      [&]() {return !g_buffer.empty(); });

    // if there are values in the queue process them
    while (!g_done && !g_buffer.empty())
    {
      std::unique_lock<std::mutex> locker(g_lockprint);
      std::cout
        << "[consumed]: " << g_buffer.front()
        << std::endl;
      g_buffer.pop();
    }
  }
}
```

The consumer thread does the following:

- Loops until it is signaled that the process of producing data is finished.
- Acquires a unique lock on the `mutex` associated with the condition variable.
- Uses the `wait_for()` overload that takes a predicate, checking that the buffer is not empty when a wakeup occurs (to avoid spurious wakeups). This method uses a timeout of 1 second and returns after the timeout has occurred, even if the condition is signaled.
- Consumes all of the data from the queue after it is signaled through the condition variable.

To test this, we start several producing threads and one consuming thread. Producer threads generate random data and, therefore, share the pseudo-random generator engines and distributions. All of this is shown in the following code sample:

```
auto seed_data = std::array<int, std::mt19937::state_size> {};
std::random_device rd {};
std::generate(std::begin(seed_data), std::end(seed_data),
              std::ref(rd));
std::seed_seq seq(std::begin(seed_data), std::end(seed_data));
auto generator = std::mt19937{ seq };
auto dsleep = std::uniform_int_distribution<>{ 100, 500 };
auto dcode = std::uniform_int_distribution<>{ 1, 99 };

std::cout << "start producing and consuming..." << std::endl;

std::thread consumerthread(consumer);
std::vector<std::thread> threads;
for (int i = 0; i < 5; ++i)
{
  threads.emplace_back(producer,
                       i + 1,
                       std::ref(generator),
                       std::ref(dsleep),
                       std::ref(dcode));
}

// work for the workers to finish
for (auto& t : threads)
  t.join();

// notify the logger to finish and wait for it
g_done = true;
```

```
    consumerthread.join();

    std::cout << "done producing and consuming" << std::endl;
```

A possible output of this program is as follows:

```
start producing and consuming...
[produced(5)]: 550
[consumed]: 550
[produced(5)]: 529
[consumed]: 529
[produced(5)]: 537
[consumed]: 537
[produced(1)]: 122
[produced(2)]: 224
[produced(3)]: 326
[produced(4)]: 458
[consumed]: 122
[consumed]: 224
[consumed]: 326
[consumed]: 458
...
done producing and consuming
```

See also

- *Working with threads*
- *Synchronizing access to shared data with mutexes and locks*

Using promises and futures to return values from threads

In the first recipe of this chapter, we discussed how to work with threads. You also learned that thread functions cannot return values, and threads should use other means, such as shared data, to do so; however, for this, synchronization is required. An alternative to communicating a return value or an exception with either the main or another thread is using `std::promise`. This recipe will explain how this mechanism works.

Getting ready

The `promise` and `future` classes used in this recipe are available in the `std` namespace in the `<future>` header.

How to do it…

To communicate a value from one thread to another through promises and futures, do this:

1. Make a promise available to the thread function through a parameter, for example:

```
void produce_value(std::promise<int>& p)
{
  // simulate long running operation
  {
    using namespace std::chrono_literals;
    std::this_thread::sleep_for(2s);
  }

  // continued at 2.
}
```

2. Call `set_value()` on the premise to set the result to represent a value or `set_exception()` to set the result to indicate an exception:

```
// continued from 1.
p.set_value(42);
```

3. Make the future associated with the premise available to the other thread function through a parameter, for example:

```
void consume_value(std::future<int>& f)
{
  // continued at 4.
}
```

4. Call `get()` on the `future` object to get the result set to the promise:

```
// continued from 3.
auto value = f.get();
```

5. In the calling thread, use `get_future()` on the promise to get the `future` associated with the promise:

```
std::promise<int> p;
std::thread t1(produce_value, std::ref(p));

std::future<int> f = p.get_future();
std::thread t2(consume_value, std::ref(f));

t1.join();
t2.join();
```

How it works…

The promise-future pair is basically a communication channel that enables a thread to communicate a value or exception with another thread through a shared state. The `promise` is an asynchronous provider of the result and has an associated `future` that represents an asynchronous return object. To establish this channel, you must first create a promise. This, in turn, creates the shared state that can be later read through the future associated with the promise.

To set a result to a promise, you can use any of the following methods:

- The `set_value()` or `set_value_at_thread_exit()` method is used to set a return value; the later function stores the value in the shared state but only makes it available through the associated future if the thread exits.
- The `set_exception()` or `set_exception_at_thread_exit()` method is used to set an exception as a return value. The exception is wrapped in an `std::exception_ptr` object. The later function stores the exception into the shared state but only makes it available when the thread exits.

To retrieve the `future` object associated with `promise`, use the `get_future()` method. To get the value from the `future` value, use the `get()` method. This blocks the calling thread until the value from the shared state is being made available. The future class has several methods for blocking the thread until the result from the shared state is made available:

- `wait()` only returns when the result is available.
- `wait_for()` returns either when the result is available or when the specified timeout expires.
- `wait_until()` returns either when the result is available or when the specified time point is reached.

If an exception is set to the `promise` value, calling the `get()` method on the future will throw this exception. The example from the previous section is rewritten as the following code to throw an exception instead of setting a result. The call to `get()` is put in a `try...catch` block, and if an exception is caught, its message is printed to the console:

```
void produce_value(std::promise<int>& p)
{
  // simulate long running operation
  {
    using namespace std::chrono_literals;
    std::this_thread::sleep_for(2s);
  }

  try
  {
    throw std::runtime_error("an error has occurred!");
  }
  catch(...)
  {
    p.set_exception(std::current_exception());
  }
}
```

```
void consume_value(std::future<int>& f)
{
  std::lock_guard<std::mutex> lock(g_mutex);
  try
  {
    std::cout << f.get() << std::endl;
  }
  catch(std::exception const & e)
  {
    std::cout << e.what() << std::endl;
  }
}
```

There’s more…

Establishing a promise-future channel in this manner is a rather explicit operation that can be avoided by using the `std::async()` function; this is a higher-level utility that runs a function asynchronously, creates an internal promise and a shared state, and returns a future associated with the shared state. We will see how `std::async()` works in the next recipe, *Executing functions asynchronously*.

See also

- *Working with threads*
- *Handling exceptions from thread functions*

Executing functions asynchronously

Threads enable us to run multiple functions at the same time; this helps us take advantage of the hardware facilities in multiprocessor or multicore systems. However, threads require explicit lower-level operations. An alternative to threads is tasks, which are units of work that run in a particular thread. The C++ standard does not provide a complete task library, but it enables developers to execute functions asynchronously on different threads and communicate results back through a promise-future channel, as seen in the previous recipe. In this recipe, we will see how to do this using `std::async()` and `std::future`.

Getting ready

We will use futures, so read the previous recipe to get a quick overview of how they work. Both `async()` and `future` are available in the `std` namespace in the `<future>` header.

For the examples in this recipe, we will use the following functions:

```
void do_something()
{
  // simulate long running operation
  {
    using namespace std::chrono_literals;
    std::this_thread::sleep_for(2s);
  }

  std::lock_guard<std::mutex> lock(g_mutex);
  std::cout << "operation 1 done" << std::endl;
}

void do_something_else()
{
  // simulate long running operation
  {
    using namespace std::chrono_literals;
    std::this_thread::sleep_for(1s);
  }

  std::lock_guard<std::mutex> lock(g_mutex);
  std::cout << "operation 2 done" << std::endl;
}

int compute_something()
{
  // simulate long running operation
  {
    using namespace std::chrono_literals;
    std::this_thread::sleep_for(2s);
  }

  return 42;
}

int compute_something_else()
{
  // simulate long running operation
  {
    using namespace std::chrono_literals;
    std::this_thread::sleep_for(1s);
  }

  return 24;
}
```

How to do it…

To execute a function asynchronously on another thread when the current thread is continuing with the execution without expecting a result do the following:

1. Use `std::async()` to start a new thread to execute the specified function. Create an asynchronous provider and return a `future` associated with it. Use the `std::launch::async` policy for the first argument to the function in order to make sure the function will run asynchronously:

```
auto f = std::async(std::launch::async, do_something);
```

2. Continue with the execution of the current thread:

```
do_something_else();
```

3. Call the `wait()` method on the `future` object returned by `std::async()` when you need to make sure the asynchronous operation is completed:

```
f.wait();
```

To execute a function asynchronously on a worker thread while the current thread continues its execution until the result from the asynchronous function is needed in the current thread, do the following:

1. Use `std::async()` to start a new thread to execute the specified function, create an asynchronous provider, and return a `future` associated with it. Use the `std::launch::async` policy of the first argument to the function to make sure the function does run asynchronously:

```
auto f = std::async(std::launch::async, compute_something);
```

2. Continue the execution of the current thread:

```
auto value = compute_something_else();
```

3. Call the `get()` method on the `future` object returned by `std::async()` when you need to get the result from the function executed asynchronously:

```
value += f.get();
```

How it works…

`std::async()` is a variadic function template that has two overloads: one that specifies a launch policy as the first argument and another that does not. The other arguments to `std::async()` are the function to execute and its arguments, if any. The launch policy is defined by a scoped enumeration called `std::launch`, available in the `<future>` header:

```
enum class launch : /* unspecified */
{
  async = /* unspecified */,
  deferred = /* unspecified */,
  /* implementation-defined */
};
```

The two available launch policies specify the following:

- With `async`, a new thread is launched to execute the task asynchronously.
- With `deferred`, the task is executed on the calling thread the first time its result is requested.

When both the flags are specified (`std::launch::async | std::launch::deferred`), it is an implementation decision whether to run the task asynchronously on a new thread or synchronously on the current thread. This is the behavior of the `std::async()` overload; it does not specify a launch policy. This behavior is not deterministic.

Do not use the non-deterministic overload of `std::async()` to run tasks asynchronously. Always use the overload that requires a launch policy, and always use `std::launch::async`.

Both the overloads of `std::async()` return a `future` object that refers to the shared state created internally by `std::async()` for the promise-future channel it establishes. When you need the result of the asynchronous operation, call the `get()` method on the future. This blocks the current thread until either the result value or an exception is made available. If the future does not transport any value or if you are not actually interested in that value but want to make sure the asynchronous operation would be completed at some point, use the `wait()` method; it blocks the current thread until the shared state is made available through the future.

The future class has two more waiting methods: `wait_for()` specifies a duration after which the call ends and returns even if the shared state is not yet available through the future; `wait_until()` specifies a time point after which the call returns even if the shared state is not yet available. These methods could be used to create a polling routine and display a status message to the user, as shown in the following example:

```
auto f = std::async(std::launch::async, do_something);

while(true)
{
  using namespace std::chrono_literals;
  auto status = f.wait_for(500ms);

  if(status == std::future_status::ready)
    break;
```

```
      std::cout << "waiting..." << std::endl;
    }

    std::cout << "done!" << std::endl;
```

The result of running this program is as follows:

```
waiting...
waiting...
waiting...
operation 1 done
done!
```

See also

- *Using promises and futures to return values from threads*

Using atomic types

The thread library provides support for managing threads and synchronizing access to shared data with mutex and locks. The standard library provides support for the complementary, lower-level atomic operations on data, that is, indivisible operations that can be executed concurrently from different threads on shared data without the risk of producing race conditions and without the use of locks. The support it provides includes atomic types, atomic operations, and memory synchronization ordering. In this recipe, we will see how to use some of these types and functions.

Getting ready

All the atomic types and operations are defined in the `std` namespace in the `<atomic>` header.

How to do it…

The following is a series of typical operations that use atomic types:

- Use the `std::atomic` class template to create atomic objects that support atomic operations, such as loading, storing, or performing arithmetic or bitwise operations:

```
std::atomic<int> counter {0};

std::vector<std::thread> threads;
for(int i = 0; i < 10; ++i)
{
  threads.emplace_back([&counter](){
    for(int i = 0; i < 10; ++i)
      ++counter;
  });
}

for(auto & t : threads) t.join();

std::cout << counter << std::endl; // 100
```

- Use the `std::atomic_flag` class for an atomic Boolean type:

```
std::atomic_flag lock = ATOMIC_FLAG_INIT;
int counter = 0;
std::vector<std::thread> threads;

for(int i = 0; i < 10; ++i)
{
  threads.emplace_back([&](){
    while(lock.test_and_set(std::memory_order_acquire));
    ++counter;
    lock.clear(std::memory_order_release);
  });
}

for(auto & t : threads) t.join();

std::cout << counter << std::endl; // 10
```

- Use the atomic type's members—`load()`, `store()`, and `exchange()`— or non-member functions—`atomic_load()`/`atomic_load_explicit()`, `atomic_store()`/`atomic_store_explicit()`, and `atomic_exchange()`/`atomic_exchange_explicit()`—to atomically read, set, or exchange the value of an atomic object.
- Use its member functions `fetch_add()` and `fetch_sub()` or non-member functions `atomic_fetch_add()`/`atomic_fetch_add_explicit()` and `atomic_fetch_sub()`/`atomic_fetch_sub_explicit()` to atomically add or subtract a value to an atomic object and return its value before the operation:

```
std::atomic<int> sum {0};
std::vector<int> numbers = generate_random();
size_t size = numbers.size();
std::vector<std::thread> threads;

for(int i = 0; i < 10; ++i)
{
  threads.emplace_back([&sum, &numbers](size_t const start,
                                        size_t const end) {
    for(size_t i = start; i < end; ++i)
    {
      std::atomic_fetch_add_explicit(
```

```
            &sum, numbers[i],
            std::memory_order_acquire);

          // same as
          // sum.fetch_add(numbers[i], std::memory_order_acquire);
        }},
        i*(size/10),
        (i+1)*(size/10));
    }

    for(auto & t : threads) t.join();
```

- Use its member functions `fetch_and()`, `fetch_or()`, and `fetch_xor()` or non-member functions `atomic_fetch_and()`/`atomic_fetch_and_explicit()`, `atomic_fetch_or()`/`atomic_fetch_or_explicit()`, and `atomic_fetch_xor()`/`atomic_fetch_xor_explicit()` to perform AND, OR, and XOR atomic operations, respectively, with the specified argument and return the value of the atomic object before the operation.
- Use the `std::atomic_flag` member functions `test_and_set()` and `clear()` or non-member functions `atomic_flag_test_and_set()`/`atomic_flag_test_and_set_explicit()` and `atomic_flag_clear()`/`atomic_flag_clear_explicit()` to set or reset an atomic flag.

How it works…

`std::atomic` is a class template that defines (including its specializations) an atomic type. The behavior of the objects of atomic types is well defined when one thread writes to the object and the other reads data, without using locks to protect access. The `std::atomic` class provides several specializations:

- Full specialization for `bool`, with a typedef called `atomic_bool`.
- Full specialization for all integral types, with typedefs called `atomic_int`, `atomic_long`, `atomic_char`, `atomic_wchar`, and many others.
- Partial specialization for pointer types.

The `atomic` class template has various member functions that perform atomic operations such as the following:

- `load()` to atomically load and return the value of the object.
- `store()` to atomically store a non-atomic value into the object; this function does not return anything.
- `exchange()` to atomically store a non-atomic value in the object and return the previous value.
- `operator=` that has the same effect as `store(arg)`.
- `fetch_add()` to atomically add a non-atomic argument to the atomic value and return the value stored previously.
- `fetch_sub()` to atomically subtract a non-atomic argument from the atomic value and return the value stored previously.
- `fetch_and()`, `fetch_or()`, and `fetch_xor()` to atomically perform a bitwise AND, OR, or XOR operation between the argument and the atomic value; store the new value in the atomic object; and return the previous value.
- Prefixing and postfixing `operator++` and `operator--` to atomically increment and decrement the value of the atomic object with 1. These operations are equivalent to using `fetch_add()` or `fetch_sub()`.
- `operator +=`, `-=`, `&=`, `|=`, and `^=` to add, subtract, or perform bitwise AND, OR, or XOR operations between the argument and the atomic value and store the new value in the atomic object. These operations are equivalent to using `fetch_add()`, `fetch_sub()`, `fetch_and()`, `fetch_or()`, and `fetch_xor()`.

Say you have an atomic variable, such as `std::atomic<int> a`; the following is not an atomic operation:

```
    a = a + 42;
```

This involves a series of operations, some of which are atomic:

- Atomically load the value of the atomic object.
- Add 42 to the value that was loaded.

- Atomically store the result in the atomic object `a`.

On the other hand, the following operation, which uses the member `operator +=`, is atomic:

```
a += 42;
```

This operation has the same effect as either of the following:

```
a.fetch_add(42);               // using member function
std::atomic_fetch_add(&a, 42); // using non-member function
```

Though `std::atomic` has full specialization for type `bool`, called `std::atomic<bool>`, the standard defines yet another atomic type called `std::atomic_flag`, which is guaranteed to be lock-free. This atomic type, however, is very different than `std::atomic_bool`, and it has only two member functions:

- `test_and_set()` that atomically sets the value to `true` and returns the previous value.
- `clear()` that atomically sets the value to `false`.

All member functions mentioned earlier, for both `std::atomic` and `std::atomic_flag`, have non-member equivalents that are prefixed with `atomic_` or `atomic_flag_`, depending on the type they refer to. For instance, the equivalent of `std::atomic::fetch_add()` is `std::atomic_fetch_add()`, and the first argument of these non-member functions is always a pointer to an `std::atomic` object. Internally, the non-member function calls the equivalent member function on the provided `std::atomic` argument. Similarly, the equivalent of `std::atomic_flag::test_and_set()` is `std::atomic_flag_test_and_set()`, and its first parameter is a pointer to an `std::atomic_flag` object.

All these member functions of `std::atomic` and `std::atomic_flag` have two sets of overloads; one of them has an extra argument representing a memory order. Similarly, all non-member functions—such as `std::atomic_load()`, `std::atomic_fetch_add()`, and `std::atomic_flag_test_and_set()`—have a companion with the suffix `_explicit`—`std::atomic_load_explicit()`, `std::atomic_fetch_add_explicit()`, and `std::atomic_flag_test_and_set_explicit()`; these functions have an extra argument that represents the memory order.

The memory order specifies how non-atomic memory accesses are to be ordered around atomic operations. By default, the memory order of all atomic types and operations is sequential consistency. Additional ordering types are defined in the `std::memory_order` enumeration and can be passed as an argument to the member functions of `std::atomic` and `std::atomic_flag` or the non-member functions with the suffix `_explicit()`.

Sequential consistency is a consistency model that requires that in a multiprocessor system, all instructions are executed in some order and all writes become instantly visible throughout the system. This model was first proposed by Leslie Lamport in the 70's, and is described as follows: "the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

Various types of memory ordering functions are described in the following table, taken from the C++ reference website (http://en.cppreference.com/w/cpp/atomic/memory_order). The details of how each one of these works is beyond the scope of this book and can be looked up in the standard C++ reference (see the link we just came across):

Model	Explanation
`memory_order_relaxed`	This is a relaxed operation. There are no synchronization or ordering constraints; only atomicity is required from this operation.
`memory_order_consume`	A load operation with this memory order performs a consume operation on the affected memory location; no reads or writes in the current thread that are dependent on the value currently loaded can be reordered before this load operation. Writes to data-dependent variables in other threads that release the same atomic variable are visible in the current thread. On most platforms, this affects compiler optimizations only.
`memory_order_acquire`	A load operation with this memory order performs the acquire operation on the affected memory location; no reads or writes in the current thread can be reordered before this load. All writes in other threads that release the same atomic variable are visible in the current thread.
`memory_order_release`	A store operation with this memory order performs the release operation; no reads or writes in the current thread can be reordered after this store. All writes in the current thread are visible in other threads that acquire the same atomic variable and writes that carry a dependency to the atomic variable become visible in other threads that consume the same atomic.
`memory_order_acq_rel`	A read-modify-write operation with this memory order is both an acquire operation and a release operation. No memory reads or writes in the current thread can be reordered before or after this store. All writes in other threads that release the same atomic variable are visible before the modification, and the modification is visible in other threads that acquire the same atomic variable.
`memory_order_seq_cst`	Any operation with this memory order is both an acquire operation and a release operation; a single total order exists in which all threads observe all modifications in the same order.

The first example in the *How to do it…* section shows several threads repeatedly by incrementing the counter concurrently. This example can be refined further by implementing a class to represent an atomic counter with methods such as `increment()` and `decrement()` that modify the value of the counter, and `get()` that retrieves its current value:

```
template
<typename T,
 typename I = typename std::enable_if<std::is_integral<T>
    ::value>::type>
class atomic_counter
```

```
{
  std::atomic<T> counter {0};
  public:
  T increment()
  {
    return counter.fetch_add(1);
  }

  T decrement()
  {
    return counter.fetch_sub(1);
  }

  T get()
  {
    return counter.load();
  }
};
```

With this class template, the first example can be rewritten in the following form with the same result:

```
atomic_counter<int> counter;

std::vector<std::thread> threads;
for(int i = 0; i < 10; ++i)
{
  threads.emplace_back([&counter](){
    for(int i = 0; i < 10; ++i)
      counter.increment();
  });
}

for(auto & t : threads) t.join();

std::cout << counter.get() << std::endl; // 100
```

See also

- *Working with threads*
- *Synchronizing access to shared data with mutexes and locks*
- *Executing functions asynchronously*

Implementing parallel map and fold with threads

In the Chapter 3, *Exploring Functions*, we discussed two higher-order functions: `map`, which applies a function to the elements of a range by either transforming the range or producing a new range, and `fold`, which combines the elements of a range into a single value. The various implementations we did were sequential. However, in the context of concurrency, threads, and asynchronous tasks, we can leverage the hardware and run parallel versions of these functions to speed up their execution for large ranges or when the transformation and aggregation are time-consuming. In this recipe, we will see a possible solution for implementing `map` and `fold` using threads.

Getting ready

You need to be familiar with the concept of the `map` and `fold` functions. It is recommended that you read the *Implementing higher-order functions map and fold* recipe from the Chapter 3, *Exploring Functions*. In this recipe, we will use the various thread functionalities presented in the *Working with threads* recipe. To measure the execution time of these functions and compare it with sequential alternatives, we will use the `perf_timer` class template introduced in the *Measuring function execution time with a standard clock* recipe in Chapter 6, *General Purpose Utilities*.

A parallel version of an algorithm can potentially speed up execution time, but this is not necessarily true in all circumstances. Context switching for threads and synchronized access to shared data can introduce a significant overhead. For some implementations and particular datasets this overhead could make a parallel version actually take a longer time to execute than a sequential version.

To determine the number of threads required to split the work, use the following function:

```
unsigned get_no_of_threads()
{
  return std::thread::hardware_concurrency();
}
```

How to do it…

To implement a parallel version of the `map` function, do the following:

1. Define a function template that takes the `begin` and `end` iterators to a range and a function to apply to all the elements:

```
template <typename Iter, typename F>
void parallel_map(Iter begin, Iter end, F f)
{
}
```

2. Check the size of the range. If the number of elements is smaller than a predefined threshold (for this implementation, the threshold is 10,000), execute the mapping in a sequential manner:

```
auto size = std::distance(begin, end);
if(size <= 10000)
  std::transform(begin, end, begin, std::forward<F>(f));
```

3. For larger ranges, split the work on multiple threads and let each thread map a part of the range. These parts should not overlap to avoid the need of synchronizing access to the shared data:

```
else
{
  auto no_of_threads = get_no_of_threads();
  auto part = size / no_of_threads;
  auto last = begin;
  // continued at 4. and 5.
}
```

4. Start the threads, and on each thread, run a sequential version of the mapping:

```
std::vector<std::thread> threads;
for(unsigned i = 0; i < no_of_threads; ++i)
{
  if(i == no_of_threads - 1) last = end;
  else std::advance(last, part);

  threads.emplace_back(
    [=,&f]{std::transform(begin, last,
                          begin, std::forward<F>(f));});
  begin = last;
}
```

5. Wait until all the threads have finished their execution:

```
for(auto & t : threads) t.join();
```

The preceding steps put together result in the following implementation:

```
template <typename Iter, typename F>
void parallel_map(Iter begin, Iter end, F f)
{
  auto size = std::distance(begin, end);
  if(size <= 10000)
    std::transform(begin, end, begin, std::forward<F>(f));
  else
  {
```

```
      auto no_of_threads = get_no_of_threads();
      auto part = size / no_of_threads;
      auto last = begin;

      std::vector<std::thread> threads;
      for(unsigned i = 0; i < no_of_threads; ++i)
      {
        if(i == no_of_threads - 1) last = end;
        else std::advance(last, part);

        threads.emplace_back(
          [=,&f]{std::transform(begin, last,
                                begin, std::forward<F>(f));});

        begin = last;
      }

      for(auto & t : threads) t.join();
    }
  }
```

To implement a parallel version of the left `fold` function, do the following:

1. Define a function template that takes a `begin` and `end` iterator to a range, an initial value, and a binary function to apply to the elements of the range:

```
      template <typename Iter, typename R, typename F>
      auto parallel_reduce(Iter begin, Iter end, R init, F op)
      {
      }
```

2. Check the size of the range. If the number of elements is smaller than a predefined threshold (for this implementation, it is 10,000), execute the folding in a sequential manner:

```
      auto size = std::distance(begin, end);
      if(size <= 10000)
        return std::accumulate(begin, end,
                               init, std::forward<F>(op));
```

3. For larger ranges, split the work into multiple threads and let each thread fold a part of the range. These parts should not overlap in order to avoid thread synchronization of shared data. The result can be returned through a reference passed to the thread function in order to avoid data synchronization:

```
      else
      {
        auto no_of_threads = get_no_of_threads();
        auto part = size / no_of_threads;
        auto last = begin;
        // continued with 4. and 5.
      }
```

4. Start the threads, and on each thread, execute a sequential version of the folding:

```
      std::vector<std::thread> threads;
      std::vector<R> values(no_of_threads);
      for(unsigned i = 0; i < no_of_threads; ++i)
      {
        if(i == no_of_threads - 1) last = end;
        else std::advance(last, part);

        threads.emplace_back(
          [=,&op](R& result){
            result = std::accumulate(begin, last, R{},
```

```
                                    std::forward<F>(op));},
          std::ref(values[i]));
        begin = last;
      }
```

5. Wait until all the threads have finished execution and fold partial results into the final result:

```
      for(auto & t : threads) t.join();

      return std::accumulate(std::begin(values), std::end(values),
                             init, std::forward<F>(op));
```

The steps we just put together result in the following implementation:

```
template <typename Iter, typename R, typename F>
auto parallel_reduce(Iter begin, Iter end, R init, F op)
{
  auto size = std::distance(begin, end);

  if(size <= 10000)
    return std::accumulate(begin, end, init, std::forward<F>(op));
  else
  {
    auto no_of_threads = get_no_of_threads();
    auto part = size / no_of_threads;
    auto last = begin;

    std::vector<std::thread> threads;
    std::vector<R> values(no_of_threads);
    for(unsigned i = 0; i < no_of_threads; ++i)
    {
      if(i == no_of_threads - 1) last = end;
      else std::advance(last, part);

      threads.emplace_back(
        [=,&op](R& result){
          result = std::accumulate(begin, last, R{},
                                   std::forward<F>(op));},
        std::ref(values[i]));

      begin = last;
    }

    for(auto & t : threads) t.join();

    return std::accumulate(std::begin(values), std::end(values),
                           init, std::forward<F>(op));
  }
}
```

How it works…

These parallel implementations of `map` and `fold` are similar in several aspects:

- They both fall back to a sequential version if the number of elements in the range is smaller than 10,000.
- They both start the same number of threads. These threads are determined using the static function `std::thread::hardware_concurrency()`, which returns the number of concurrent threads supported by the implementation. However, this value is rather a hint than an accurate value and should be used with that in mind.
- No shared data is used to avoid synchronization of access. Even though all the threads work on the elements from the same range, they all process parts of the range that do not overlap.
- Both these functions are implemented as function templates that take a begin and end iterator to define the range to be processed. In order to split the range into multiple parts to be processed independently by different threads, use additional iterators in the middle of the range. For this, we use `std::advance()` to increment an iterator with a particular number of positions. This works well for vectors or arrays, but is very inefficient for containers such as lists. Therefore, this implementation is suited only for ranges that have random access iterators.

The sequential version of `map` and `fold` can be simply implemented in C++ with `std::transform()` and `std::accumulate()`. In fact, to verify the correctness of the parallel algorithms and check whether they provide any execution speedup, we can compare them with the execution of these general purpose algorithms.

To put it to the test, we will use `map` and `fold` on a vector with size varying from 10,000 to 50 million elements. The range is first mapped (that is, transformed) by doubling the value of each element and then the result is folded into a single value by adding together all the elements of the range. For simplicity, each element in the range is equal to its 1-based index (the first element is 1, the second element is 2, and so on). The following sample runs both the sequential and parallel versions of map and fold on vectors of different sizes and prints the execution time in a tabular format.

As an exercise, you can vary the number of elements and also the number of threads and see how the parallel version performs compared to the sequential version.

```
std::vector<int> sizes
{
  10000, 100000, 500000,
  1000000, 2000000, 5000000,
  10000000, 25000000, 50000000
};

std::cout
  << std::right << std::setw(8) << std::setfill(' ') << "size"
  << std::right << std::setw(8) << "s map"
  << std::right << std::setw(8) << "p map"
  << std::right << std::setw(8) << "s fold"
```

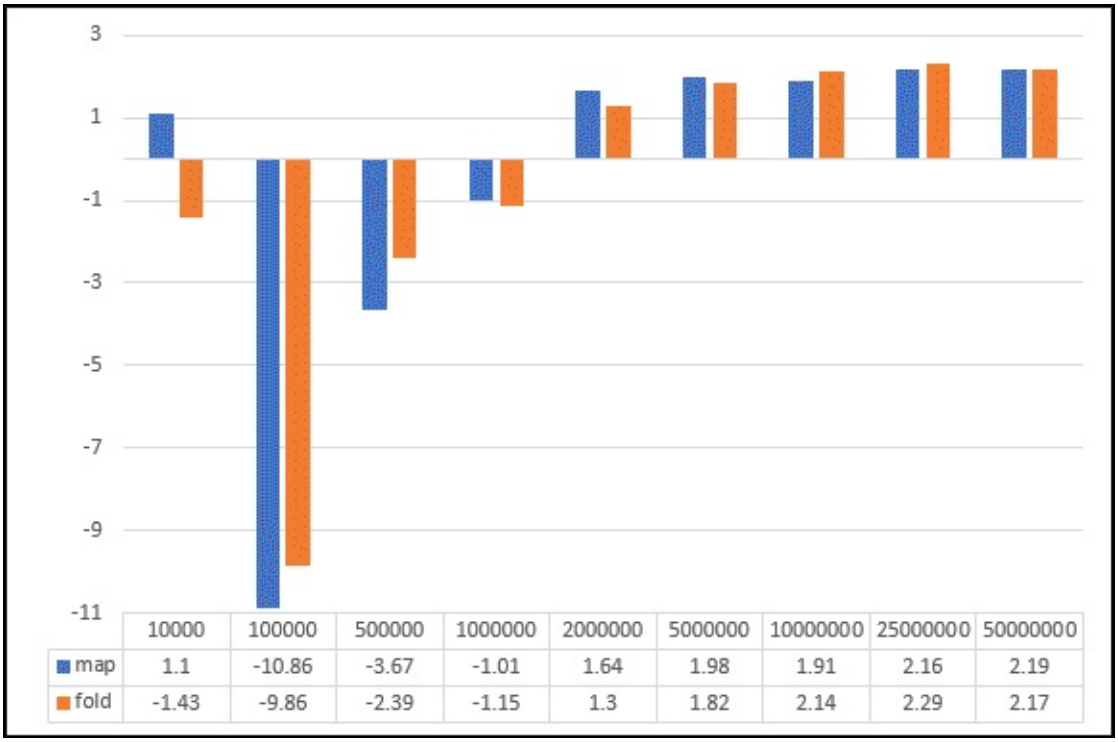
```
      << std::right << std::setw(8) << "p fold"
      << std::endl;

    for (auto const size : sizes)
    {
      std::vector<int> v(size);
      std::iota(std::begin(v), std::end(v), 1);

      auto v1 = v;
      auto s1 = 0LL;

      auto tsm = perf_timer<>::duration([&] {
        std::transform(std::begin(v1), std::end(v1), std::begin(v1),
                       [](int const i) {return i + i; }); });
      auto tsf = perf_timer<>::duration([&] {
        s1 = std::accumulate(std::begin(v1), std::end(v1), 0LL,
                             std::plus<>()); });

      auto v2 = v;
      auto s2 = 0LL;
      auto tpm = perf_timer<>::duration([&] {
        parallel_map(std::begin(v2), std::end(v2),
                     [](int const i) {return i + i; }); });
      auto tpf = perf_timer<>::duration([&] {
        s2 = parallel_reduce(std::begin(v2), std::end(v2), 0LL,
                             std::plus<>()); });

      assert(v1 == v2);
      assert(s1 == s2);

      std::cout
        << std::right << std::setw(8) << std::setfill(' ') << size
        << std::right << std::setw(8)
        << std::chrono::duration<double, td::micro>(tsm).count()
        << std::right << std::setw(8)
        << std::chrono::duration<double, std::micro>(tpm).count()
        << std::right << std::setw(8)
        << std::chrono::duration<double, std::micro>(tsf).count()
        << std::right << std::setw(8)
        << std::chrono::duration<double, std::micro>(tpf).count()
        << std::endl;
    }
```

A possible output of this program is shown in the next screenshot (executed on a machine running Windows 64-bit with an Intel Core i7 processor and 4 physical and 8 logical cores). The parallel version, especially the `fold` implementation, performs better than the sequential version. But this is true only when the length of the vector exceeds a certain size. In the following table, we can see that for up to 1 million elements, the sequential version is still faster. The parallel version executes faster when there are 2 million or more elements in the vector. Notice that the actual times vary slightly from one run to another, but they can be very different on different machines:

size	s map	p map	s fold	p fold
10000	11	10	7	10
100000	108	1573	72	710
500000	547	2006	361	862
1000000	1146	1163	749	862
2000000	2503	1527	1677	1289
5000000	5937	3000	4203	2314
10000000	11959	6269	8269	3868
25000000	29872	13823	20961	9156
50000000	60049	27457	41374	19075

To better visualize these results, we can represent the speedup of the parallel version in the form of a bar chart. In the following image, the blue bars represent the speedup of a parallel `map` implementation, and the orange bars show the speedup of the parallel `fold` implementation. A positive value indicates that the parallel version is faster; a

negative version indicates that the sequential version is faster:



	10000	100000	500000	1000000	2000000	5000000	10000000	25000000	50000000
map	1.1	-10.86	-3.67	-1.01	1.64	1.98	1.91	2.16	2.19
fold	-1.43	-9.86	-2.39	-1.15	1.3	1.82	2.14	2.29	2.17

See also

- *Implementing higher-order functions map and fold* recipe of Chapter 3, *Exploring Functions*
- *Implementing parallel map and fold with tasks*
- *Working with threads*

Implementing parallel map and fold with tasks

Tasks are a higher-level alternative to threads for performing concurrent computations. `std::async()` enables us to execute functions asynchronously, without the need to handle lower-level threading details. In this recipe, we will take the same task of implementing a parallel version of the `map` and `fold` functions, as in the previous recipe, but we will use tasks and see how it compares with the thread version.

Getting ready

The solution presented in this recipe is similar in many aspects to the one that uses threads in the previous recipe, *Implementing parallel map and fold with threads*. Make sure you read that one before continuing with the current recipe.

How to do it…

To implement a parallel version of the `map` function, do the following:

1. Define a function template that takes a begin and end iterator to a range, and a function to apply to all the elements:

```
template <typename Iter, typename F>
void parallel_map(Iter begin, Iter end, F f)
{
}
```

2. Check the size of the range. For a number of elements smaller than the predefined threshold (for this implementation, the threshold is 10,000), execute the mapping in a sequential manner:

```
auto size = std::distance(begin, end);
if(size <= 10000)
  std::transform(begin, end, begin, std::forward<F>(f));
```

3. For larger ranges, split the work into multiple tasks and let each task map a part of the range. These parts should not overlap to avoid synchronizing thread access to shared data:

```
else
{
  auto no_of_tasks = get_no_of_threads();
  auto part = size / no_of_tasks;
  auto last = begin;
  // continued at 4. and 5.
}
```

4. Start the asynchronous functions and run a sequential version of the mapping on each one of them:

```
std::vector<std::future<void>> tasks;
for(unsigned i = 0; i < no_of_tasks; ++i)
{
  if(i == no_of_tasks - 1) last = end;
  else std::advance(last, part);

  tasks.emplace_back(std::async(
    std::launch::async,
      [=,&f]{std::transform(begin, last, begin,
                            std::forward<F>(f));}));
  begin = last;
}
```

5. Wait until all the asynchronous functions have finished their execution:

```
for(auto & t : tasks) t.wait();
```

These steps put together result in the following implementation:

```
template <typename Iter, typename F>
void parallel_map(Iter begin, Iter end, F f)
{
  auto size = std::distance(begin, end);
  if(size <= 10000)
```

```
      std::transform(begin, end, begin, std::forward<F>(f));
    else
    {
      auto no_of_tasks = get_no_of_threads();
      auto part = size / no_of_tasks;
      auto last = begin;

      std::vector<std::future<void>> tasks;
      for(unsigned i = 0; i < no_of_tasks; ++i)
      {
        if(i == no_of_tasks - 1) last = end;
        else std::advance(last, part);

        tasks.emplace_back(std::async(
          std::launch::async,
            [=,&f]{std::transform(begin, last, begin,
                                  std::forward<F>(f));}));

        begin = last;
      }

      for(auto & t : tasks) t.wait();
    }
  }
```

To implement a parallel version of the left `fold` function, do the following:

1. Define a function template that takes a begin and end iterator to a range, an initial value, and a binary function to apply to the elements of the range:

```
      template <typename Iter, typename R, typename F>
      auto parallel_reduce(Iter begin, Iter end, R init, F op)
      {
      }
```

2. Check the size of the range. For a number of elements smaller than the predefined threshold (for this implementation, the threshold is 10,000), execute the folding in a sequential manner:

```
      auto size = std::distance(begin, end);
      if(size <= 10000)
        return std::accumulate(begin, end, init,
                               std::forward<F>(op));
```

3. For larger ranges, split the work into multiple tasks and let each task fold a part of the range. These parts should not overlap to avoid synchronizing thread access to the shared data. The result can be returned through a reference passed to the asynchronous function to avoid synchronization:

```
      else
      {
        auto no_of_tasks = get_no_of_threads();
        auto part = size / no_of_tasks;
        auto last = begin;
        // continued at 4. and 5.
      }
```

4. Start the asynchronous functions and execute a sequential version of the folding on each one of them:

```
      std::vector<std::future<R>> tasks;
      for(unsigned i = 0; i < no_of_tasks; ++i)
      {
        if(i == no_of_tasks - 1) last = end;
```

```
        else std::advance(last, part);

        tasks.emplace_back(
          std::async(
            std::launch::async,
            [=,&op]{return std::accumulate(
                             begin, last, R{},
                             std::forward<F>(op));}));
        begin = last;
      }
```

5. Wait until all the asynchronous functions have finished execution and fold the partial results into the final result:

```
      std::vector<R> values;
      for(auto & t : tasks)
        values.push_back(t.get());

      return std::accumulate(std::begin(values), std::end(values),
                             init, std::forward<F>(op));
```

These steps put together result in the following implementation:

```
template <typename Iter, typename R, typename F>
auto parallel_reduce(Iter begin, Iter end, R init, F op)
{
  auto size = std::distance(begin, end);

  if(size <= 10000)
    return std::accumulate(begin, end, init, std::forward<F>(op));
  else
  {
    auto no_of_tasks = get_no_of_threads();
    auto part = size / no_of_tasks;
    auto last = begin;

    std::vector<std::future<R>> tasks;
    for(unsigned i = 0; i < no_of_tasks; ++i)
    {
      if(i == no_of_tasks - 1) last = end;
      else std::advance(last, part);

      tasks.emplace_back(
        std::async(
          std::launch::async,
          [=,&op]{return std::accumulate(
                           begin, last, R{},
                           std::forward<F>(op));}));

      begin = last;
    }

    std::vector<R> values;
    for(auto & t : tasks)
      values.push_back(t.get());

    return std::accumulate(std::begin(values), std::end(values),
                           init, std::forward<F>(op));
  }
}
```

How it works…

The implementation just proposed is only slightly different than what we did in the previous recipe. Threads were replaced with asynchronous functions, starting with `std::async()`, and results were made available through the returned `std::future`. The number of asynchronous functions that are launched concurrently is equal to the number of threads the implementation can support. This is returned by the static method `std::thread::hardware_concurrency()`, but this value is only a hint and should not be considered very reliable.

There are mainly two reasons for taking this approach:

- Seeing how a function implemented for parallel execution with threads can be modified to use asynchronous functions and, therefore, avoid lower-level details of threading.
- Running a number of asynchronous functions equal to the number of supported threads can potentially run one function per thread; this could provide the fastest execution time for the parallel function because there is a minimum overhead of context switching and waiting time.

We can test the performance of the new `map` and `fold` implementations using the same method as in the previous recipe:

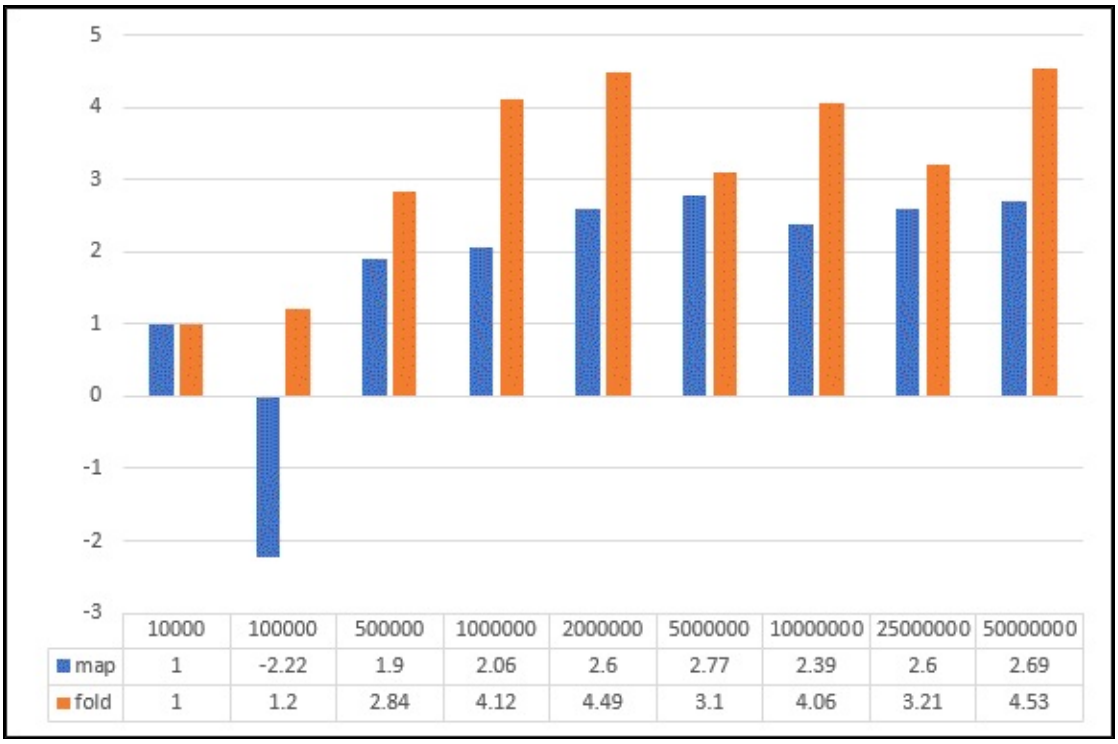
```
std::vector<int> sizes
{
  10000, 100000, 500000,
  1000000, 2000000, 5000000,
  10000000, 25000000, 50000000
};

std::cout
 << std::right << std::setw(8) << std::setfill(' ') << "size"
 << std::right << std::setw(8) << "s map"
 << std::right << std::setw(8) << "p map"
 << std::right << std::setw(8) << "s fold"
 << std::right << std::setw(8) << "p fold"
 << std::endl;

for(auto const size : sizes)
{
  std::vector<int> v(size);
  std::iota(std::begin(v), std::end(v), 1);

  auto v1 = v;
  auto s1 = 0LL;

  auto tsm = perf_timer<>::duration([&] {
    std::transform(std::begin(v1), std::end(v1), std::begin(v1),
                   [](int const i) {return i + i; }); });
  auto tsf = perf_timer<>::duration([&] {
    s1 = std::accumulate(std::begin(v1), std::end(v1), 0LL,
                         std::plus<>()); });

auto v2 = v;
auto s2 = 0LL;
auto tpm = perf_timer<>::duration([&] {
  parallel_map(std::begin(v2), std::end(v2),
               [](int const i) {return i + i; }); });
auto tpf = perf_timer<>::duration([&] {
  s2 = parallel_reduce(std::begin(v2), std::end(v2), 0LL,
```

```
                          std::plus<>()); });

    assert(v1 == v2);
    assert(s1 == s2);

    std::cout
      << std::right << std::setw(8) << std::setfill(' ') << size
      << std::right << std::setw(8)
      << std::chrono::duration<double, std::micro>(tsm).count()
      << std::right << std::setw(8)
      << std::chrono::duration<double, std::micro>(tpm).count()
      << std::right << std::setw(8)
      << std::chrono::duration<double, std::micro>(tsf).count()
      << std::right << std::setw(8)
      << std::chrono::duration<double, std::micro>(tpf).count()
      << std::endl;
    }
```

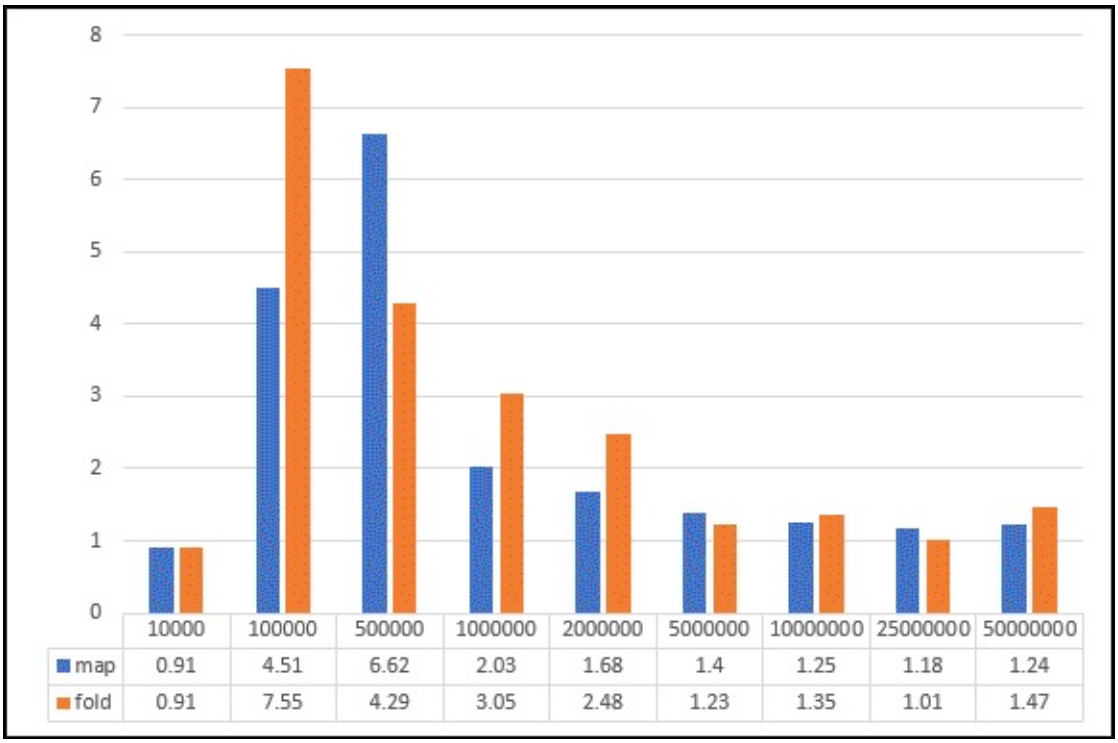
A possible output of the preceding program, which can vary slightly from one execution to another and greatly from one machine to another, is as follows:

```
    size   s map   p map  s fold  p fold
   10000      11      11      11      11
  100000     117     260     113      94
  500000     576     303     571     201
 1000000    1180     573    1165     283
 2000000    2371     911    2330     519
 5000000    5942    2144    5841    1886
10000000   11954    4999   11643    2871
25000000   30525   11737   29053    9048
50000000   59665   22216   58689   12942
```

Similar to the illustration of the threads solution, the speedup of the parallel `map` and `fold` implementations can be seen in the following chart. Negative values indicate that the sequential version was faster:



If we compare this with the results from the parallel version using threads, we will find that these are faster execution times and the speedup is significant, especially for the `fold` function. The following chart shows the speedup of tasks' implementation over threads' implementation. In this chart, a value smaller than 1 means that the threads implementation was faster:



	10000	100000	500000	1000000	2000000	5000000	10000000	25000000	50000000
map	0.91	4.51	6.62	2.03	1.68	1.4	1.25	1.18	1.24
fold	0.91	7.55	4.29	3.05	2.48	1.23	1.35	1.01	1.47

There’s more…

The implementation shown earlier is only one of the possible approaches we can take for parallelizing the `map` and `fold` functions. A possible alternative uses the following strategy:

- Divide the range to process into two equal parts.
- Recursively call the parallel function asynchronously to process the first part of the range.
- Recursively call the parallel function synchronously to process the second part of the range.
- After the synchronous recursive call is finished, wait for the asynchronous recursive call to end too before finishing the execution.

This divide-and-conquer algorithm can potentially create a lot of tasks. Depending on the size of the range, the number of asynchronous calls can greatly exceed the number of threads, and in this case, there will be lots of waiting time that will affect the overall execution time. So the following alternative is available:

```
template <typename Iter, typename F>
void parallel_map(Iter begin, Iter end, F f)
{
  auto size = std::distance(begin, end);

  if(size <= 10000)
  {
    std::transform(begin, end, begin, std::forward<F>(f));
  }
  else
  {
    auto middle = begin;
    std::advance(middle, size / 2);

    auto result = std::async(
      std::launch::deferred,
      parallel_map<Iter, F>,
      begin, middle, std::forward<F>(f));
    parallel_map(middle, end, std::forward<F>(f));
    result.wait();
  }
}

template <typename Iter, typename R, typename F>
auto parallel_reduce(Iter begin, Iter end, R init, F op)
{
  auto size = std::distance(begin, end);

  if(size <= 10000)
    return std::accumulate(begin, end, init, std::forward<F>(op));
  else
  {
    auto middle = begin;
    std::advance(middle, size / 2);

    auto result1 = std::async(
      std::launch::async,
      parallel_reduce<Iter, R, F>,
      begin, middle, R{}, std::forward<F>(op));
    auto result2 = parallel_reduce(middle, end, init,
                                   std::forward<F>(op));
    return result1.get() + result2;
  }
}
```

When we compare the execution time with the first implementation using asynchronous functions—that is, using the same testing method—we see that this version (indicated by `p2` in the next output) is similar to the sequential version for both `map` and `fold` and much worse than the first parallel version shown earlier (indicated by `p1`):

```
    size   s map p1 map  p2 map  s fold p1 fold p2 fold
   10000      11     11      10       7      10      10
  100000     111    275     120      72      96     426
  500000     551    230     596     365     210    1802
 1000000    1142    381    1209     753     303    2378
 2000000    2411    981    2488    1679     503    4190
 5000000    5962   2191    6237    4177    1969    7974
10000000   11961   4517   12581    8384    2966   15174
```

See also

- *Implementing parallel map and fold with threads*
- *Executing functions asynchronously*

Robustness and Performance

This chapter includes the following recipes:

- Using exceptions for error handling
- Using noexcept for functions that do not throw
- Ensuring constant correctness for a program
- Creating compile-time constant expressions
- Performing correct type casts
- Using unique_ptr to uniquely own a memory resource
- Using shared_ptr to share a memory resource
- Implementing move semantics

Introduction

C++ is often the first choice when it comes to selecting an object-oriented programming language with performance and flexibility as key goals. Modern C++ provides language and library features, such as rvalue references, move semantics, and smart pointers. When combined with good practices for exception handling, constant correctness, type-safe conversions, resource allocation and releasing, they enable developers to write better, more robust and performant code. This chapter contains recipes which address all of these topics.

Using exceptions for error handling

Exceptions are responses to exceptional circumstances that can appear when a program is running. They enable the transfer of the control flow to another part of the program. Exceptions are a mechanism for simpler and more robust error handling, as opposed to returning error codes that could greatly complicate and clutter the code. In this recipe, we will look at some key aspects related to throwing and handling exceptions.

Getting ready

I assume you have basic knowledge of the mechanism of throwing and catching exceptions.

How to do it…

Use the following practices to deal with exceptions:

- Throw exceptions by value:

```
void throwing_func()
{
  throw std::system_error(
    std::make_error_code(std::errc::timed_out));
}
```

- Catch exceptions by reference, or in most cases, by constant reference:

```
try
{
  throwing_func();
}
catch (std::exception const & e)
{
  std::cout << e.what() << std::endl;
}
```

- Order catch statements from the most derived class to the base class of the hierarchy when catching multiple exceptions from a class hierarchy:

```
auto exprint = [](std::exception const & e)
{
  std::cout << e.what() << std::endl;
};

try
{
  throwing_func();
}
catch (std::system_error const & e)
{
  exprint(e);
}
catch (std::runtime_error const & e)
{
  exprint(e);
}
catch (std::exception const & e)
{
  exprint(e);
}
```

- Use `catch(...)` to catch all the exceptions, regardless of their type:

```
try
{
  throwing_func();
}
catch (std::exception const & e)
{
  std::cout << e.what() << std::endl;
}
catch (...)
{
  std::cout << "unknown exception" << std::endl;
}
```

- Use `throw;` to rethrow the current exception. This can be used to create a single

exception handling function for multiple exceptions. Throw the exception object (for example, `throw e;`) when you want to hide the original location of the exception:

```
void handle_exception()
{
  try
  {
    throw; // throw current exception
  }
  catch (const std::logic_error & e)
  { /* ... */ }
  catch (const std::runtime_error & e)
  { /* ... */ }
  catch (const std::exception & e)
  { /* ... */ }
}

try
{
  throwing_func();
}
catch (...)
{
  handle_exception();
}
```

How it works…

Most functions have to indicate the success or failure of their execution. This can be achieved in different ways. Here's a possibility: return an error code (with a special value for success) to indicate the specific reason for failure. A variation of this is to return a Boolean value to only indicate success or failure. Another alternative is to return invalid objects or null pointers. In any case, the return value from the functions should be checked. This can lead to complex, cluttered, hard to read and maintain real-world code. Moreover, the process of checking the return value of a function is always executed, regardless of whether the function was successful or failed. On the other hand, exceptions are thrown and handled only when a function fails, which should happen more rarely than successful executions. This can actually lead to faster code than code that returns and tests error codes.

Exceptions and error codes are not mutually exclusive. Exceptions should be used only for transferring the control flow in exceptional situations, not for controlling the data flow in a program.

Class constructors are special functions that do not return any value. They are supposed to construct an object, but in the case of failure, they will not be able to indicate this with a return value. Exceptions should be a mechanism which constructors should use to indicate failure. Together with the *resource acquisition is initialization (RAII)* idiom, this ensures safe acquisition and release of resources in all situations. On the other hand, exceptions are not allowed to leave a destructor. When this happens, the program abnormally terminates with a call to `std::terminate()`. This is the case for destructors called during stack unwinding due to the occurring of another exception. When an exception occurs, the stack is unwound from the point where the exception was thrown to the block where the exception is handled, and this process involves the destruction of all local objects in all those stack frames. If the destructor of an object that is being destroyed during this process throws an exception, another stack unwinding process should begin, which conflicts with the one already under way. Because of this, the program terminates abnormally.

The rule of thumb for dealing with exceptions in constructors and destructors is as follows:
1. Use exceptions to indicate the errors that occur in constructors.
2. Do not throw or let exceptions leave destructors.

It is possible to throw any type of exception. However, in most cases, you should throw temporaries and catch exceptions by constant reference. The following are some guidelines for exception throwing:

- Prefer throwing either standard exceptions or your own exceptions derived from `std::exception` or another standard exception.
- Avoid throwing exceptions of built-in types, such as integers.
- When using a library or framework that provides its own exception hierarchy, prefer throwing exceptions from this hierarchy or your own exceptions derived from it, at

least in the parts of the code tightly related to it to keep the code consistent.

There’s more…

As mentioned in the preceding section, when you need to create your own exception types, derive them from one of the standard exceptions that are available, unless you are using a library or framework with its own exception hierarchy. The C++ standard defines several categories of exceptions that need to be considered for this purpose:

- The `std::logic_error` represents an exception that indicates an error in the program logic, such as an invalid argument, an index beyond the bounds of a range, and so on. There are various standard derived classes, such as `std::invalid_argument`, `std::out_of_range`, and `std::length_error`.
- The `std::runtime_error` represents an exception that indicates an error beyond the scope of the program or that cannot be predicted due to various factors, including external ones, such as overflows and underflows or operating system errors. The C++ standard also provides several derived classes from `std::runtime_error`, including `std::overflow_error`, `std::underflow_error`, or `std::system_error`.
- Exceptions prefixed with `bad_`, such as `std::bad_alloc`, `std::bad_cast`, and `std::bad_function_call`, represent various errors in a program, such as failure to allocate memory, failure to dynamically cast or make a function call, and so on.

The base class for all these exceptions is `std::exception`. It has a non-throwing virtual method called `what()` that returns a pointer to an array of characters representing the description of the error. When you need to derive custom exceptions from a standard exception, use the appropriate category, such as logical or runtime error. If none of these categories is suitable, then you can derive directly from `std::exception`. The following is a list of possible solutions you can use to derive from a standard exception:

- If you need to derive from `std::exception`, then override the virtual method `what()` to provide a description of the error:

```
class simple_error : public std::exception
{
public:
  virtual const char* what() const noexcept override
  {
    return "simple exception";
  }
};
```

- If you derive from `std::logic_error` or `std::runtime_error` and you only need to provide a static description that does not depend on runtime data, then pass the description text to the base class constructor:

```
class another_logic_error : public std::logic_error
{
public:
  another_logic_error():
    std::logic_error("simple logic exception")
  {}
};
```

- If you derive from `std::logic_error` or `std::runtime_error` but the description message depends on runtime data, provide a constructor with parameters and use them to build the description message. You can either pass the description message to the base class constructor or return it from the overridden `what()` method:

```
class advanced_error : public std::runtime_error
{
  int error_code;
  std::string make_message(int const e)
  {
    std::stringstream ss;
    ss << "error with code " << e;
    return ss.str();
  }
public:
  advanced_error(int const e) :
  std::runtime_error(make_message(e).c_str()),error_code(e)
  {
  }

  int error() const noexcept
  {
    return error_code;
  }
};
```

See also

- *Handling exceptions from thread functions* from Chapter 8, *Leveraging Threading and Concurrency*
- *Using noexcept for functions that do not throw*

Using noexcept for functions that do not throw

Exception specification is a language feature that can enable performance improvements, but on the other hand, when done incorrectly, it can abnormally terminate the program. The exception specification from C++03 which allowed you to indicate what types of exceptions a function could throw has been deprecated and replaced with the new C++11 `noexcept` specification. This specification only allows you to indicate whether a function may throw or not. This recipe provides information about the modern exception specifications in C++, as well as guidelines on when to use it.

How to do it…

Use the following constructs to specify or query exception specifications:

- Use `nothrow` in a function declaration to indicate that the function is not throwing any exception:

```
void func_no_throw() noexcept
{
}
```

- Use `nothrow(expr)` in a function declaration, such as template metaprogramming, to indicate that the function may or may not throw an exception based on a condition that evaluates to `bool`:

```
template <typename T>
T generic_func_1()
noexcept(std::is_nothrow_constructible<T>::value)
{
  return T{};
}
```

- Use the `noexcept` operator at compile time to check whether an expression is declared to not throw any exception:

```
template <typename T>
T generic_func_2() noexcept(noexcept(T{}))
{
  return T{};
}

template <typename F, typename A>
auto func(F&& f, A&& arg) noexcept
{
  static_assert(!noexcept(f(arg)), "F is throwing!");
  return f(arg);
}

std::cout << noexcept(func_no_throw) << std::endl;
```

How it works…

As of C++17, exception specification is part of the function type, but not part of the function signature; it may appear as part of any function declarator. Because exception specification is not part of the function signature, two function signatures cannot differ only in the exception specification. Prior to C++17, exception specification was not part of the function type and could only appear as part of lambda declarators or top-level function declarators; they could not appear even in `typedef` or type alias declarations. Further discussions on exception specification refer solely to the C++17 standard.

There are several ways in which the process of throwing an exception can be specified:

- If no exception specification is present, then the function could potentially throw exceptions.
- `noexcept(false)` is equivalent to no exception specification.
- `noexcept(true)` and `noexcept` indicate that a function does not throw any exception.
- `throw()` is equivalent to `noexcept(true)` but deprecated.

Using exception specifications must be done with care because, if an exception (either thrown directly or from another function that is called) leaves a function marked as non-throwing, the program is terminated immediately and abnormally with a call to *`std::terminate()`*.

Pointers to the functions that do not throw exceptions can be implicitly converted into pointers to functions that may throw exceptions, but not the opposite. On the other hand, if a virtual function has a non-throwing exception specification, it indicates that all the declarations of all the overrides must preserve this specification unless an overridden function is declared as deleted.

At compile time, it is possible to check whether a function is declared to be non-throwing or not using `operator noexcept`. This operator takes an expression and returns `true` if the expression is declared as either non-throwing or `false`. It does not evaluate the expression it checks. The `noexcept` operator, along with the `noexcept` specifier, is particularly useful in template metaprogramming to indicate whether a function may throw exceptions for some types. It is also used with `static_assert` declarations to check whether an expression breaks the non-throwing guarantee of a function, as seen in the examples in the *How to do it…* section. The following code provides more examples of how the `noexcept` operator works:

```
int double_it(int const i) noexcept
{
  return i + i;
}

int half_it(int const i)
{
  throw std::runtime_error("not implemented!");
}

struct foo
{
  foo() {}
```

```
};
std::cout << std::boolalpha
  << noexcept(func_no_throw()) << std::endl               // true
  << noexcept(generic_func_1<int>()) << std::endl         // true
  << noexcept(generic_func_1<std::string>()) << std::endl // true
  << noexcept(generic_func_2<int>()) << std::endl         // true
  << noexcept(generic_func_2<std::string>()) << std::endl // true
  << noexcept(generic_func_2<foo>()) << std::endl         // false
  << noexcept(double_it(42)) << std::endl                 // true
  << noexcept(half_it(42)) << std::endl                   // false
  << noexcept(func(double_it, 42)) << std::endl           // true
  << noexcept(func(half_it, 42)) << std::endl;            // true
```

There’s more…

As mentioned earlier, a function declared with the `noexcept` specifier that exits due to an exception causes the program to terminate abnormally. Therefore, the `noexcept` specifier should be used with caution. Its presence can enable code optimizations, which help increase performance while preserving the strong exception guarantee. An example of this is library containers.

The strong exception guarantee specifies that either an operation is completed successfully, or is completed with an exception that leaves the program in the same state it was before the operation started. This ensures commit-or-rollback semantics.

Many standard containers provide some of their operations with a strong exception guarantee. An example is vector’s `push_back()` method. This method could be optimized by using the move constructor or move assignment operator instead of the copy constructor or copy assignment operator of the vector’s element type. However, in order to preserve its strong exception guarantee, this can only be done if the move constructor or assignment operator does not throw exceptions. If they do, then the copy constructor or assignment operator must be used. The `std::move_if_noexcept()` utility function does this if the move constructor of its type argument is marked with `noexcept`. The ability to indicate that move constructors or move assignment operators do not throw is probably the most important scenario where `noexcept` is used.

Consider the following rules for exception specification:

- If a function could potentially throw an exception, then do not use any exception specifier.
- Mark only those functions with `noexcept` that are guaranteed would not to throw an exception.
- Mark only those functions with `noexcept(expression)` that could potentially throw exceptions based on a condition.
- Do not mark a function with either `noexcept` or `noexcept(expression)` unless it provides a direct real benefit.

See also

- *Using exceptions for error handling*

Ensuring constant correctness for a program

Although there is no formal definition, constant correctness means objects that are not supposed to be modified (are immutable) remain unmodified indeed. As a developer, you can enforce this by using the `const` keyword for declaring parameters, variables, and member functions. In this recipe, we will explore the benefits of constant correctness and how to achieve it.

How to do it…

To ensure constant correctness for a program, you should always declare as constant:

- Parameters to functions that are not supposed to be modified within the function:

```
struct session {};

session connect(std::string const & uri,
                int const timeout = 2000)
{
   /* do something */
   return session { /* ... */ };
}
```

- Class data members that do not change:

```
class user_settings
{
public:
  int const min_update_interval = 15;
  /* other members */
};
```

- Class member functions that do not modify the object state as seen from the outside:

```
class user_settings
{
  bool show_online;
  public:
  bool can_show_online() const {return show_online;}
  /* other members */
};
```

- Function locals whose value do not change throughout their lifetime:

```
user_settings get_user_settings()
{
  return user_settings {};
}

void update()
{
  user_settings const us = get_user_settings();
  if(us.can_show_online()) { /* do something */ }
  /* do more */
}
```

How it works…

Declaring objects and member functions constant has several important benefits:

- You prevent both accidental and intentional changes of the object which, in some cases, can result in incorrect program behavior.
- You enable the compiler to perform better optimizations.
- You document the semantics of the code for other users.

Constant correctness is not a matter of personal style but a core principle that should guide C++ development.

Unfortunately, the importance of constant correctness has not been, and is still not, stressed enough in books, C++ communities, and working environments. But the rule of thumb is that everything that is not supposed to change should be declared constant. This should be done all the time and not only at later stages of development when you might need to clean up and refactor the code.

When you declare a parameter or variable as constant, you can either put the `const` keyword before the type (`const T c`) or after the type (`T const c`). These two are equivalent, but regardless of which of the two styles you use, reading of the declaration must be done from the right-hand side. `const T c` is read as *c is a T that is constant* and `T const c` as *c is a constant T*. This gets a little bit more complicated with pointers. The following table presents various pointer declarations and their meanings:

Expression	**Description**
`T* p`	P is a non-constant pointer to a non-constant T
`const T* p`	P is a non-constant pointer to a T that is constant
`T const * p`	P is a non-constant pointer to a constant T (same as above)
`const T * const p`	P is a constant pointer to a T that is constant
`T const * const p`	P is a constant pointer to a constant T (same as above)
`T** p`	P is a non-constant pointer to a non-constant pointer to a non-constant T
`const T** p`	P is a non-constant pointer to a non-constant pointer to a constant T
`T const ** p`	Same as `T const ** p`
`const T* const * p`	P is a non-constant pointer to a constant pointer, which is a constant T
`T const * const * p`	Same as `T const * const * p`

Placing the `const` keyword after the type is more natural because it is consistent with the reading direction, from right to left. For this reason, all

the examples in this book use this style.

When it comes to references, the situation is similar: `const T & c` and `T const & c` are equivalent, which means *c is a reference to a constant T*. However, `T const & const c`, which would mean that *c is a constant reference to a constant T*, does not make sense because references—aliases of a variable—are implicitly constant in the sense that they cannot be modified to represent an alias to another variable.

A non-constant pointer to a non-constant object, that is, `T*`, can be implicitly converted into a non-constant pointer to a constant object, `T const *`. However, `T**` cannot be implicitly converted into `T const **` (which is the same with `const T**`). It is because this could lead to constant objects being modified through a pointer to a non-constant object, as shown in the following example:

```
int const c = 42;
int* x;
int const ** p = &x; // this is an actual error
*p = &c;
*x = 0;              // this modifies c
```

If an object is constant, only the constant functions of its class can be invoked. However, declaring a member function constant does not mean that the function can only be called on constant objects; it could also mean that the function does not modify the state of the object as seen from the outside. This is a key aspect, but it is usually misunderstood. A class has an internal state that it can expose to its clients through its public interface. However, not all the internal states might be exposed, and what is visible from the public interface might not have a direct representation in the internal state. (If you model order lines and have the item quantity and item selling price fields in the internal representation, then you might have a public method that exposes the order line amount by multiplying quantity by the price.) Therefore, the state of an object, as visible from its public interface, is a logical state. Defining a method as constant is a statement that the function does not alter the logical state. However, the compiler prevents you from modifying data members using such methods. To avoid this problem, data members that are supposed to be modified from constant methods should be declared mutable.

In the following example, `computation` is a class with the `compute()` method, which performs a long-running computation operation. Because it does not affect the logical state of the object, this function is declared constant. However, to avoid computing the result of the same input again, the computed values are stored in a cache. To be able to modify the cache from the constant function, it is declared `mutable`:

```
class computation
{
  double compute_value(double const input) const
  {
    /* long running operation */
    return input;
  }

  mutable std::map<double, double> cache;
public:
  double compute(double const input) const
  {
    auto it = cache.find(input);
    if(it != cache.end()) return it->second;
```

```
      auto result = compute_value(input);
      cache[input] = result;

      return result;
    }
  };
```

A similar situation is represented by the following class that implements a thread-safe container. Access to shared internal data is protected with `mutex`. The class provides methods such as adding and removing values, and also methods such as `contains()` which indicate whether an item exists in the container. Because this member function is not intended to modify the logical state of the object, it is declared constant. However, access to the shared internal state must be protected with the mutex. In order to lock and unlock the mutex, both mutable operations, the mutex must be declared `mutable`:

```
template <typename T>
class container
{
  std::vector<T> data;
  mutable std::mutex mutex;
public:
  void add(T const & value)
  {
    std::lock_guard<std::mutex> lock(mutex);
    data.push_back(value);
  }

  bool contains(T const & value) const
  {
    std::lock_guard<std::mutex> lock(mutex);
    return std::find(std::begin(data), std::end(data), value)
           != std::end(data);
  }
};
```

Sometimes, a method or an operator is overloaded to have both constant and non-constant versions. This is often the case with the subscript operator or methods that provide direct access to the internal state. The reason for this is that the method is supposed to be available for both constant and non-constant objects. The behavior should be different, though: for non-constant objects, the method should allow the client to modify the data it provides access to, but for constant objects, it should not. Therefore, the non-constant subscript operator returns a reference to a non-constant object, and the constant subscript operator returns a reference to a constant object:

```
class contact {};

class addressbook
{
  std::vector<contact> contacts;
public:
  contact& operator[](size_t const index);
  contact const & operator[](size_t const index) const;
};
```

It should be noted that, if a member function is constant, even if an object is constant, data returned by this member function may not be constant.

There’s more…

The `const` qualifier of an object can be removed with a `const_cast` conversion, but this should only be used when you know that the object was not declared constant. You can read more about this in the *Performing correct type casts* recipe.

See also

- *Creating compile-time constant expressions*
- *Performing correct type casts*

Creating compile-time constant expressions

The possibility to evaluate expressions at compile time improves runtime execution because there is less code to run and the compiler can perform additional optimizations. Compile-time constants can be not only literals (such as a number or string), but also the result of a function execution. If all the input values of a function (regardless of whether they are arguments, locals, or globals) are known at compile time, the compiler can execute the function and have the result available at compile time. This is what generalized the constant expressions introduced in C++11, which were relaxed in C++14. The keyword `constexpr` (short for *constant expression*) can be used to declare compile-time constant objects and functions. We have seen this in several examples in the previous chapters. Now it’s time to learn how it actually works.

Getting ready

The way generalized constant expressions work has been relaxed in C++14, but this introduced some breaking changes to C++11. For instance, in C++11 a `constexpr` function was implicitly `const`, but this is no longer the case in C++14. In this recipe, we will discuss generalized constant expressions as defined in C++14.

How to do it…

Use the `constexpr` keyword when you want to:

- Define non-member functions that can be evaluated at compile time:

```
constexpr unsigned int factorial(unsigned int const n)
{
  return n > 1 ? n * factorial(n-1) : 1;
}
```

- Define constructors that can be executed at compile time to initialize `constexpr` objects and member functions to be invoked during this period:

```
class point3d
{
  double const x_;
  double const y_;
  double const z_;
public:
  constexpr point3d(double const x = 0,
                    double const y = 0,
                    double const z = 0)
    :x_{x}, y_{y}, z_{z}
  {}

  constexpr double get_x() const {return x_;}
  constexpr double get_y() const {return y_;}
  constexpr double get_z() const {return z_;}
};
```

- Define variables that can have their values evaluated at compile time:

```
constexpr unsigned int size = factorial(6);
char buffer[size] {0};
constexpr point3d p {0, 1, 2};
constexpr auto x = p.get_x();
```

How it works…

The `const` keyword is used for declaring variables as constant at runtime; this means that, once initialized, they cannot be changed. However, evaluating the constant expression may still imply runtime computation. The `constexpr` keyword is used for declaring variables that are constant at compile time or functions that can be executed at compile time. `constexpr` functions and objects can replace macros and hardcoded literals without any performance penalty.

Declaring a function as `constexpr` does not mean that it is always evaluated at compile time. It only enables the use of the function in expressions that are evaluated during compile time. This only happens if all the input values of the function can be evaluated at compile time. However, the function may also be invoked at runtime. The following code shows two invocations of the same function, first at compile time, and second at runtime:

```
constexpr unsigned int size = factorial(6); // compile time evaluation

int n;
std::cin >> n;
auto result = factorial(n);                 // runtime evaluation
```

There are some restrictions in regard to where `constexpr` can be used:

- In the case of variables, it is used only when:
  - Its type is a literal type
  - It is initialized upon declaration
  - The expression used for initializing the variable is a constant expression
- In the case of functions, it is used only when:
  - It is not virtual
  - The return type and the type of parameters are all literal types
  - There is at least one set of arguments for which the invocation of the function would produce a constant expression
  - The function body must be either deleted or defaulted; it should not contain `asm` declarations, `goto` statements, labels, `try...catch` blocks, and local variables that are either not initialized, of non-literal types, or with static or thread storage duration
  - If the function is a defaulted copy or move assignment operator, then the class must not contain any mutable variant members
- In the case of constructors, it is used only when:
  - All the parameters are of a literal type.
  - There is no virtual base class for the class.
  - It does not contain a function try block.
  - The function body must be either deleted, defaulted, or satisfy several additional conditions. The compound statement must satisfy all the conditions we just mentioned for regular functions. All the constructors that initialize non-static data members, including base classes, must also be `constexpr`. And all non-static

data members must be initialized by the constructor (for union types, only one of the variants needs to be initialized).
- If it is a defaulted copy or move constructor, then the class must not contain any mutable variant members.

A function that is `constexpr` is not implicitly `const` (as of C++14), so you need to explicitly use the `const` specifier if the function does not alter the logical state of the object. However, a function that is `constexpr` is implicitly `inline`. On the other hand, an object that is declared `constexpr` is implicitly `const`. The following two declarations are equivalent:

```
constexpr const unsigned int size = factorial(6);
constexpr unsigned int size = factorial(6);
```

There are situations when you may need to use both `constexpr` and `const` in a declaration, as they would refer to different parts of the declaration. In the following example, `p` is a `constexpr` pointer to a constant integer:

```
static constexpr int c = 42;
constexpr int const * p = &c;
```

Reference variables can also be `constexpr` if, and only if, they alias an object with static storage duration or a function:

```
static constexpr int const & r = c;
```

See also

- *Ensuring constant correctness for a program*

Performing correct type casts

It is often the case that data has to be converted from one type to another type. Some conversions are necessary at compile time (such as `double` to `int`); others are necessary at runtime (such as upcasting and downcasting pointers to the classes in a hierarchy). The language supports compatibility with C casting style in either the `(type)expression` or `type(expression)` form. However, this type of casting breaks the type safety of C++. Therefore, the language also provides several conversions, `static_cast`, `dynamic_cast`, `const_cast`, and `reinterpret_cast`. They are used to better indicate intent and write safer code. In this recipe, we look at how these casts can be used.

How to do it…

Use the following casts to perform type conversions:

- Use `static_cast` to perform type casting of non-polymorphic types, including casting of integers to enumerations, from floating point to integral values or from a pointer type to another pointer type, such as from a base class to a derived class (downcasting) or from a derived class to a base class (upcasting), but without any runtime checks:

```
enum options {one = 1, two, three};

int value = 1;
options op = static_cast<options>(value);

int x = 42, y = 13;
double d = static_cast<double>(x) / y;

int n = static_cast<int>(d);
```

- Use `dynamic_cast` to perform type casting of pointers or references of polymorphic types from a base class to a derived class or the other way around. These checks are performed at runtime and require that *Runtime Type Information (RTTI)* is enabled:

```
struct base
{
  virtual void run() {}
  virtual ~base() {}
};

struct derived : public base
{
};

derived d;
base b;

base* pb = dynamic_cast<base*>(&d);         // OK
derived* pd = dynamic_cast<derived*>(&b);   // fail

try
{
  base& rb = dynamic_cast<base&>(d);       // OK
  derived& rd = dynamic_cast<derived&>(b); // fail
}
catch (std::bad_cast const & e)
{
  std::cout << e.what() << std::endl;
}
```

- Use `const_cast` to perform conversion between types with different `const` and `volatile` specifiers, such as removing `const` from an object that was not declared as `const`:

```
void old_api(char* str, unsigned int size)
{
  // do something without changing the string
}

std::string str{"sample"};
old_api(const_cast<char*>(str.c_str()),
        static_cast<unsigned int>(str.size()));
```

- Use `reinterpret_cast` to perform a bit reinterpretation, such as conversion between integers and pointer types, from pointer types to integer, from a pointer type to any other pointer type, without involving any runtime checks:

```
class widget
{
public:
  typedef size_t data_type;

  void set_data(data_type d) { data = d; }
  data_type get_data() const { return data; }
private:
  data_type data;
};

widget w;
user_data* ud = new user_data();
// write
w.set_data(reinterpret_cast<widget::data_type>(ud));
// read
user_data* ud2 = reinterpret_cast<user_data*>(w.get_data());
```

How it works…

The explicit type conversion, sometimes referred to as C-style casting or static casting, is a legacy of compatibility of C++ with the C language and it enables you to perform various conversions:

- Between arithmetical types
- Between pointer types
- Between integral and pointer types
- Between const or volatile qualified and unqualified types
- Any combination of the previous one and any of the preceding conversions

This type of casting does not work well with polymorphic types or in templates. Because of this, C++ provides the four casts we saw in the examples earlier. Using these casts leads to several important benefits:

- They express user intent better, both to the compiler and others that read the code.
- They enable safer conversion between various types (except for `reinterpret_cast`).
- They can be easily searched in the source code.

`static_cast` is not a direct equivalent of C casting, even though the name might suggest that. This cast is performed at compile time and can be used to perform implicit conversions, the reverse of implicit conversions, and conversion from pointers to types from a hierarchy of classes. It cannot be used to trigger a conversion between unrelated pointer types, though. In the following example, converting from `int*` to `double*` using `static_cast` is a compiler error. However, converting from `base*` to `derived*` (where `base` and `derived` are the classes shown in the *How to do it…* section) does not produce any compiler error but a runtime error, when trying to use the newly obtained pointer. On the other hand, `static_cast` cannot be used to remove `const` and `volatile` qualifiers:

```
int* pi = new int{ 42 };
double* pd = static_cast<double*>(pi);   // compiler error

base b;
derived* pd = static_cast<derived*>(&b); // compilers OK, runtime error
base* pb1 = static_cast<base*>(&d);      // OK

int const c = 42;
int* pc = static_cast<int*>(&c);         // compiler error
```

Safe typecasting of expressions up, down, or sideways along an inheritance hierarchy can be performed with `dynamic_cast`. This cast is performed at runtime and requires that RTTI is enabled. Because of this, it incurs a runtime overhead. Dynamic casting can only be used for pointers and references. When `dynamic_cast` is used to convert an expression to a pointer type and the operation fails, the result is a null pointer. When it is used to convert an expression to a reference type and the operation fails, an `std::bad_cast` exception is thrown. Therefore, always put a `dynamic_cast` conversion to a reference type within a `try...catch` block.

RTTI is a mechanism which exposes information about object data types at runtime. This is available only for polymorphic types (types that have at least one virtual method, including a virtual destructor, which all base classes should have). RTTI is usually an optional compiler feature (or might not be supported at all), which means using this functionality may require using a compiler switch.

Though dynamic casting is performed at runtime, if you attempt to convert it between non-polymorphic types, you'll get a compiler error:

```
struct struct1 {};
struct struct2 {};

struct1 s1;
struct2* ps2 = dynamic_cast<struct2*>(&s1); // compiler error
```

`reinterpret_cast` is more like a compiler directive. It does not translate into any CPU instructions, but only instructs the compiler to interpret the binary representation of an expression as it was of another, specified type. This is a type-unsafe conversion and should be used with care. It can be used to convert expression between integral types and pointers, pointer types, and function pointer types. Because no checks are done, `reinterpret_cast` can be successfully used to convert expressions between unrelated types, such as from `int*` to `double*`, which produces undefined behavior:

```
int* pi = new int{ 42 };
double* pd = reinterpret_cast<double*>(pi);
```

A typical use of `reinterpret_cast` is to convert expressions between types in code that uses operating-system or vendor-specific APIs. Many APIs store user data in the form of a pointer or an integral type. Therefore, if you need to pass the address of a user-defined type to such APIs, you need to convert values of unrelated pointer types or a pointer type value to an integral type value. A similar example was provided in the previous section, where `widget` was a class which stored user-defined data in a data member and provided methods for accessing it: `set_data()` and `get_data()`. If you need to store a pointer to an object in `widget`, then use `reinterpret_cast`, as shown in this example.

`const_cast` is similar to `reinterpret_cast` in the sense that it is a compiler directive and does not translate into CPU instructions. It is used to cast away `const` or `volatile` qualifiers, an operation which none of the other three conversions discussed here can do.

`const_cast` should only be used to remove `const` or `volatile` qualifiers when the object is not declared `const` or `volatile`. Anything else incurs undefined behavior, as shown in the example below:

```
int const a = 42;
int const * p = &a;
int* q = const_cast<int*>(p);
*q = 0; // undefined behavior
```

There’s more…

When using explicit type conversion in the form `(type)expression`, be aware that it would select the first choice from the following list, which satisfies specific casts requirements:

1. `const_cast<type>(expression)`
2. `static_cast<type>(expression)`
3. `static_cast<type>(expression)` + `const_cast<type>(expression)`
4. `reinterpret_cast<type>(expression)`
5. `reinterpret_cast<type>(expression)` + `const_cast<type>(expression)`

Moreover, unlike the specific C++ casts, C-style cast can be used to convert between incomplete class types. If both `type` and `expression` are pointers to incomplete types, then it is not specified whether `static_cast` or `reinterpret_cast` is selected.

See also

- *Ensuring constant correctness for a program*

Using unique_ptr to uniquely own a memory resource

Manual handling of heap memory allocation and releasing is one of the most controversial features of C++. All allocations must be properly paired with a corresponding delete operation in the correct scope. If the scope of memory allocation is a function, for instance, and memory needs to be released before the function returns, then this has to happen on all the return paths, including the abnormal situation where a function returns because of an exception. C++11 features, such as rvalues and move semantics, have enabled the development of smart pointers; these pointers can manage a memory resource and automatically release it when the smart pointer is destroyed. In this recipe, we will look at `std::unique_ptr`: a smart pointer that owns and manages another object or an array of objects allocated on the heap and performs the disposal operation when the smart pointer goes out of scope.

Getting ready

For this recipe, you need to be familiar with move semantics and the `std::move()` conversion function. The `unique_ptr` class is available in the `std` namespace in the `<memory>` header.

For simplicity and readability, we will not use in this recipe the fully qualified names `std::unique_ptr` and `std::shared_ptr` but `unique_ptr` and `shared_ptr`.

In the following examples, we will use the ensuing class:

```
class foo
{
  int a;
  double b;
  std::string c;
public:
  foo(int const a = 0, double const b = 0, std::string const & c = "")
  :a(a), b(b), c(c)
  {}

  void print() const
  {
    std::cout << '(' << a << ',' << b << ',' << std::quoted(c) << ')'
              << std::endl;
  }
};
```

How to do it…

The following is a list of typical operations you need to be aware of for working with `unique_ptr`:

- Use the available overloaded constructors to create an `unique_ptr` that manages objects or an array of objects through a pointer. The default constructor creates a pointer that does not manage any object:

```
std::unique_ptr<int>   pnull;
std::unique_ptr<int>   pi(new int(42));
std::unique_ptr<int[]> pa(new int[3]{ 1,2,3 });
std::unique_ptr<foo>   pf(new foo(42, 42.0, "42"));
```

- Alternatively, use the `std::make_unique()` function template, available in C++14, to create `unique_ptr` objects:

```
std::unique_ptr<int>   pi = std::make_unique<int>(42);
std::unique_ptr<int[]> pa = std::make_unique<int[]>(3);
std::unique_ptr<foo>   pf = std::make_unique<foo>(42, 42.0, "42");
```

- Use the overloaded constructor that takes a custom deleter if the default delete operator is not appropriate for destroying the managed object or array:

```
struct foo_deleter
{
  void operator()(foo* pf) const
  {
    std::cout << "deleting foo..." << std::endl;
    delete pf;
  }
};

std::unique_ptr<foo, foo_deleter> pf(new foo(42, 42.0, "42"),
                                     foo_deleter());
```

- Use `std::move()` to transfer the ownership of an object from one `unique_ptr` to another:

```
auto pi = std::make_unique<int>(42);
auto qi = std::move(pi);
assert(pi.get() == nullptr);
assert(qi.get() != nullptr);
```

- To access the raw pointer to the managed object, use `get()` if you want to retain ownership of the object or `release()` if you want to release the ownership as well:

```
void func(int* ptr)
{
  if (ptr != nullptr)
    std::cout << *ptr << std::endl;
  else
    std::cout << "null" << std::endl;
}

std::unique_ptr<int> pi;
func(pi.get()); // prints null

pi = std::make_unique<int>(42);
func(pi.get()); // prints 42
```

- Dereference the pointer to the managed object using `operator*` and `operator->`:

```
auto pi = std::make_unique<int>(42);
*pi = 21;

auto pf = std::make_unique<foo>();
pf->print();
```

- If a `unique_ptr` manages an array of objects, `operator[]` can be used to access individual elements of the array:

```
std::unique_ptr<int[]> pa = std::make_unique<int[]>(3);
for (int i = 0; i < 3; ++i)
  pa[i] = i + 1;
```

- To check whether `unique_ptr` can manage an object or not, use the explicit `operator bool` or check whether `get() != nullptr` (which is what `operator bool` does):

```
std::unique_ptr<int> pi(new int(42));
if (pi) std::cout << "not null" << std::endl;
```

- `unique_ptr` objects can be stored in a container. Objects returned by `make_unique()` can be stored directly. A lvalue object could be statically converted into an rvalue object with `std::move()` if you want to give up the ownership of the managed object to the `unique_ptr` object in the container:

```
std::vector<std::unique_ptr<foo>> data;
for (int i = 0; i < 5; i++)
  data.push_back(
std::make_unique<foo>(i, i, std::to_string(i)));

auto pf = std::make_unique<foo>(42, 42.0, "42");
data.push_back(std::move(pf));
```

How it works…

`unique_ptr` is a smart pointer that manages an object or an array allocated on the heap through a raw pointer, performing an appropriate disposal when the smart pointer goes out of scope, is assigned a new pointer with `operator=`, or it gives up ownership using the `release()` method. By default, operator delete is used to dispose of the managed object. However, the user may supply a custom deleter when constructing the smart pointer. This deleter must be a function object, either an lvalue reference to a function object or a function, and this callable object must take a single argument of the type `unique_ptr<T, Deleter>::pointer`.

C++14 has added the `std::make_unique()` utility function template to create an `unique_ptr`. It avoids memory leaks in some particular contexts, but it has some limitations:

- It can only be used to allocate arrays; you cannot also use it to initialize them, which is possible with a `unique_ptr` constructor. The following two pieces of sample code are equivalent:

```
// allocate and initialize an array
std::unique_ptr<int[]> pa(new int[3]{ 1,2,3 });

// allocate and then initialize an array
std::unique_ptr<int[]> pa = std::make_unique<int[]>(3);
for (int i = 0; i < 3; ++i)
  pa[i] = i + 1;
```

- It cannot be used to create an `unique_ptr` object with a user-defined deleter.

As we just mentioned, the great advantage of `make_unique()` is that it helps avoiding memory leaks in some contexts when exceptions are being thrown. `make_unique()` itself can throw `std::bad_alloc` if the allocation fails or any exception thrown by the constructor of the object it creates. Let’s consider the following example:

```
void some_function(std::unique_ptr<foo> p)
{ /* do something */ }

some_function(std::unique_ptr<foo>(new foo()));
some_function(std::make_unique<foo>());
```

Regardless of what happens with the allocation and construction of `foo`, there will be no memory leaks, irrespective of whether you use `make_unique()` or the constructor of `unique_ptr`. However, this situation changes in a slightly different version of the code:

```
void some_other_function(std::unique_ptr<foo> p, int const v)
{
}

int function_that_throws()
{
  throw std::runtime_error("not implemented");
}

// possible memory leak
some_other_function(std::unique_ptr<foo>(new foo),
                    function_that_throws());

// no possible memory leak
```

```
some_other_function(std::make_unique<foo>(),
                    function_that_throws());
```

In this example, `some_other_function()` has an extra parameter: an integer value. The integer argument passed to this function is the returned value of another function. If this function call throws, using the constructor of`unique_ptr` to create the smart pointer can produce a memory leak. The reason for this is that, upon calling `some_other_function()`, the compiler might first call `foo`, then `function_that_throws()`, and then the constructor of `unique_ptr`. If `function_that_throws()` throws an error, then the allocated `foo` would leak. If the calling order is `function_that_throws()` and then `new foo()` and the constructor of `unique_ptr`, a memory leak will not happen; this is because the stack starts unwinding before the `foo` object is allocated. However, by using the `make_unique()` function, this situation is avoided. This is because the only calls made are to `make_unique()` and `function_that_throws()`. If `function_that_throws()` is called first, then the `foo` object will not be allocated at all. If `make_unique()` is called first, the `foo` object is constructed and its ownership is passed to `unique_ptr`. If a later call to `function_that_throws()` does throw, then the `unique_ptr` will be destroyed when the stack is unwound and the `foo` object will be destroyed from the smart pointer's destructor.

Constant `unique_ptr` objects cannot transfer the ownership of a managed object or array to another `unique_ptr` object. On the other hand, access to the raw pointer to the managed object can be obtained with either `get()` or `release()`. The first method only returns the underlying pointer, but the latter also releases the ownership of the managed object, hence the name. After a call to `release()`, the `unique_ptr` object will be empty and a call to `get()` will return `nullptr`.

A `unique_ptr` that manages the object of a `Derived` class can be implicitly converted into a `unique_ptr` that manages an object of class `Base` if `Derived` is derived from `Base`. This implicit conversion is safe only if `Base` has a virtual destructor (as all base classes should have); otherwise, undefined behavior is employed:

```
struct Base
{
  virtual ~Base()
  {
    std::cout << "~Base()" << std::endl;
  }
};

struct Derived : public Base
{
  virtual ~Derived()
  {
    std::cout << "~Derived()" << std::endl;
  }
};

std::unique_ptr<Derived> pd = std::make_unique<Derived>();
std::unique_ptr<Base> pb = std::move(pd);
```

`unique_ptr` can be stored in containers, such as `std::vector`. Because only one `unique_ptr` object can own the managed object at any point, the smart pointer cannot be copied to the container; it has to be moved. This is possible with `std::move()` that performs a `static_cast` to an rvalue reference type. This allows the ownership of the managed object to be transferred to the `unique_ptr` object that is created in the container.

See also

- *Using shared_ptr to share a memory resource*

Using shared_ptr to share a memory resource

Managing dynamically allocated objects or arrays with `std::unique_ptr` is not possible when the object or array has to be shared because a `std::unique_ptr` retains its sole ownership. The C++ standard provides another smart pointer, called `std::shared_ptr`; it is similar to `std::unique_ptr` in many ways, but the difference is that it can share the ownership of an object or array with other `std::shared_ptr`. In this recipe, we will see how `std::shared_ptr` works and how it differs from `std::uniqueu_ptr`. We will also look at `std::weak_ptr`, which is a non-resource-owning smart pointer that holds a reference to an object managed by a `std::shared_ptr`.

Getting ready

Make sure you read the previous recipe, *Using unique_ptr to uniquely own a memory resource*, to become familiar with how `unique_ptr` and `make_unique()` work. We will use the `foo`, `foo_deleter`, `Base`, and `Derived` classes defined in this recipe and also make several references to it.

Both the `shared_ptr` and `weak_ptr` classes, as well as the `make_shared()` function template, are available in the `std` namespace in the `<memory>` header.

For simplicity and readability, we will not use in this recipe the fully qualified names `std::unique_ptr`, `std::shared_ptr`, `std::weak_pointer` *but* `unique_ptr`, `shared_ptr` *and* `weak_ptr`.

How to do it…

The following is a list of the typical operations you need to be aware of for working with `shared_ptr` and `weak_ptr`:

- Use one of the available overloaded constructors to create a `shared_ptr` that manages an object through a pointer. The default constructor creates an empty `shared_ptr` which does not manage any object:

```
std::shared_ptr<int> pnull1;
std::shared_ptr<int> pnull2(nullptr);
std::shared_ptr<int> pi1(new int(42));
std::shared_ptr<int> pi2 = pi1;
std::shared_ptr<foo> pf1(new foo());
std::shared_ptr<foo> pf2(new foo(42, 42.0, "42"));
```

- Alternatively, use the `std::make_shared()` function template, available since C++11, for creating `shared_ptr` objects:

```
std::shared_ptr<int> pi = std::make_shared<int>(42);
std::shared_ptr<foo> pf1 = std::make_shared<foo>();
std::shared_ptr<foo> pf2 = std::make_shared<foo>(42, 42.0, "42");
```

- Use the overloaded constructor that takes a custom deleter if the default delete operation is not appropriate for destroying the managed object:

```
std::shared_ptr<foo> pf1(new foo(42, 42.0, "42"),
                         foo_deleter());
std::shared_ptr<foo> pf2(
          new foo(42, 42.0, "42"),
          [](auto p) {
  std::cout << "deleting foo from lambda..." << std::endl;
  delete p;}
);
```

- Always specify a deleter when managing an array of objects. The deleter can either be a partial specialization of `std::default_delete` for arrays or any function that takes a pointer to the template type:

```
std::shared_ptr<int> pa1(
  new int[3]{ 1, 2, 3 },
  std::default_delete<int[]>());

std::shared_ptr<int> pa2(
  new int[3]{ 1, 2, 3 },
  [](auto p) {delete[] p; });
```

- To access the raw pointer to the managed object, use the `get()` function:

```
void func(int* ptr)
{
  if (ptr != nullptr)
    std::cout << *ptr << std::endl;
  else
    std::cout << "null" << std::endl;
}

std::shared_ptr<int> pi;
func(pi.get());
```

```
pi = std::make_shared<int>(42);
func(pi.get());
```

- Dereference the pointer to the managed object using `operator*` and `operator->`:

```
std::shared_ptr<int> pi = std::make_shared<int>(42);
*pi = 21;

std::shared_ptr<foo> pf = std::make_shared<foo>(42, 42.0, "42");
pf->print();
```

- If a `shared_ptr` manages an array of objects, `operator[]` can be used to access the individual elements of the array. This is only available in C++17:

```
std::shared_ptr<int> pa1(
  new int[3]{ 1, 2, 3 },
  std::default_delete<int[]>());

for (int i = 0; i < 3; ++i)
  pa1[i] *= 2;
```

- To check whether a `shared_ptr` could manage an object or not, use the explicit `operator bool` or check whether `get() != nullptr` (which is what `operator bool` does):

```
std::shared_ptr<int> pnull;
if (pnull) std::cout << "not null" << std::endl;

std::shared_ptr<int> pi(new int(42));
if (pi) std::cout << "not null" << std::endl;
```

- `shared_ptr` objects can be stored in containers, such as `std::vector`:

```
std::vector<std::shared_ptr<foo>> data;
for (int i = 0; i < 5; i++)
  data.push_back(
    std::make_shared<foo>(i, i, std::to_string(i)));

auto pf = std::make_shared<foo>(42, 42.0, "42");
data.push_back(std::move(pf));
assert(!pf);
```

- Use `weak_ptr` to maintain a non-owning reference to a shared object, which can be later accessed through a `shared_ptr` constructed from the `weak_ptr` object:

```
auto sp1 = std::make_shared<int>(42);
assert(sp1.use_count() == 1);

std::weak_ptr<int> wpi = sp1;
assert(sp1.use_count() == 1);

auto sp2 = wpi.lock();
assert(sp1.use_count() == 2);
assert(sp2.use_count() == 2);

sp1.reset();
assert(sp1.use_count() == 0);
assert(sp2.use_count() == 1);
```

- Use the `std::enable_shared_from_this` class template as the base class for a type when you need to create `shared_ptr` objects for instances that are already managed by another `shared_ptr` object:

```
struct Apprentice;

struct Master : std::enable_shared_from_this<Master>
{
  ~Master() { std::cout << "~Master" << std::endl; }
  void take_apprentice(std::shared_ptr<Apprentice> a);
private:
  std::shared_ptr<Apprentice> apprentice;
};

struct Apprentice
{
  ~Apprentice() { std::cout << "~Apprentice" << std::endl; }
  void take_master(std::weak_ptr<Master> m);
private:
  std::weak_ptr<Master> master;
};

void Master::take_apprentice(std::shared_ptr<Apprentice> a)
{
  apprentice = a;
  apprentice->take_master(shared_from_this());
}

void Apprentice::take_master(std::weak_ptr<Master> m)
{
  master = m;
}

auto m = std::make_shared<Master>();
auto a = std::make_shared<Apprentice>();
m->take_apprentice(a);
```

How it works…

`shared_ptr` is very similar to `unique_ptr` in many aspects; however, it serves a different purpose: sharing the ownership of an object or array. Two or more `shared_ptr` smart pointers can manage the same dynamically allocated object or array, which is automatically destroyed when the last smart pointer goes out of scope, is assigned a new pointer with `operator=`, or is reset with method `reset()`. By default, the object is destroyed with `operator delete`; however, the user could supply a custom deleter to the constructor, something that is not possible using `std::make_shared()`. If the `shared_ptr` is used to manage an array of objects, a custom deleter must be supplied. In this case, you can use `std::default_delete<T[]>`, which is a partial specialization of the `std::default_delete` class template that uses `operator delete[]` to delete the dynamically allocated array.

The utility function `std::make_shared()` (available since C++11) unlike `std::make_unique()`, which is only available since C++14, should be used to create smart pointers unless you need to provide a custom deleter. The primary reason for this is the same as for `make_unique()`: avoiding potential memory leaks in some contexts when an exception is thrown. For more information on this, read the explanation provided on `std::make_unique()` in the previous recipe.

Also, as in the case of `unique_ptr`, a `shared_ptr` that manages an object of a `Derived` class can be implicitly converted into a `shared_ptr` that manages an object of the `Base` class. This is possible if the `Derived` class is derived from `Base`. This implicit conversion is safe only if `Base` has a virtual destructor (as all the base classes should have when objects are supposed to be deleted polymorphically through a pointer or reference to the base class); otherwise, undefined behavior is employed. In C++17, several new non-member functions have been added: `std::static_pointer_cast()`, `std::dynamic_pointer_cast()`, `std::const_pointer_cast()`, and `std::reinterpret_pointer_cast()`. These apply `static_cast`, `dynamic_cast`, `const_cast`, and `reinterpret_cast` to the stored pointer, returning a new `shared_ptr` to the designated type. In the following example, `Base` and `Derived` are the same classes used in the previous recipe:

```
std::shared_ptr<Derived> pd = std::make_shared<Derived>();
std::shared_ptr<Base> pb = pd;

std::static_pointer_cast<Derived>(pb)->print();
```

There are situations when you need a smart pointer for a shared object but without it contributing to the shared ownership. Suppose you model a tree structure where a node has references to its children and they are represented by `shared_ptr` objects. On the other hand, say a node needs to keep a reference to its parent. If this reference were also `shared_ptr`, then it would create circular references and no object would ever be automatically destroyed.

`weak_ptr` is a smart pointer used to break such circular dependencies. It holds a non-owning reference to an object or array managed by a `shared_ptr`. The `weak_ptr` can be created from a `shared_ptr` object. In order to access the managed object, you need to get a temporary `shared_ptr` object. To do so, we need to use the `lock()` method. This method atomically

checks whether the referred object still exists and returns either an empty `shared_ptr`, if the object no longer exists, or a `shared_ptr` that owns the object, if it still exists. Because `weak_ptr` is a non-owning smart pointer, the referred object can be destroyed before `weak_ptr` goes out of scope or when all the owning `shared_ptr` objects have been destroyed, reset, or assigned to other pointers. The method `expired()` can be used to check whether the referenced object has been destroyed or is still available.

In the *How to do it…* section, the preceding example models a master-apprentice relationship. There is a `Master` class and an `Apprentice` class. The `Master` class has a reference to an `Apprentice` class and a method called `take_apprentice()` to set the `Apprentice` object. The `Apprentice` class has a reference to a `Master` class and the method `take_master()` to set the `Master` object. In order to avoid circular dependencies, one of these references must be represented by a `weak_ptr`. In the proposed example, the `Master` class had a `shared_ptr` to own the `Apprentice` object, and the `Apprentice` class had a `weak_ptr` to track a reference to the `Master` object. This example, however, is a bit more complex because here, the `Apprentice::take_master()` method is called from `Master::take_apprentice()` and needs a `weak_ptr<Master>`. In order to call it from within the `Master` class, we must be able to create a `shared_ptr<Master>` in the `Master` class, using the `this` pointer. The only way to do that in a safe manner is to use `std::enable_shared_from_this`.

`std::enable_shared_from_this` is a class template that must be used as a base class for all the classes where you need to create a `shared_ptr` for the current object (the `this` pointer) when this object is already managed by another `shared_ptr`. Its type template parameter must be the class that derives from it, as in the curiously recurring template pattern. It has two methods: `shared_from_this()` returns a `shared_ptr`, which shares the ownership of the `this` object, and `weak_from_this()` returns a `weak_ptr`, which shares a non-owning reference to the `this` object. The latter method is only available in C++17. These methods can be called only on an object that is managed by an existing `shared_ptr`; otherwise, they throw an `std::bad_weak_ptr` exception, as of C++17. Prior to C++17, the behavior was undefined.

Not using `std::enable_shared_from_this` and creating a `shared_ptr<T>(this)` directly would lead to having multiple `shared_ptr` objects which would manage the same object independently, without knowing each other. When this happens, the object ends up being destroyed multiple times from different `shared_ptr` objects.

See also

- *Using unique_ptr to uniquely own a memory resource*

Implementing move semantics

Move semantics are a key feature that drives the performance improvements of modern C++. They enable moving, rather than copying, resources or, in general, objects which are expensive to copy. However, it requires that classes implement a move constructor and assignment operator. These are provided by the compiler in some circumstances, but in practice, it is often the case that you have to explicitly write them. In this recipe, we will see how to implement the move constructor and the move assignment operator.

Getting ready

You are expected to have basic knowledge of rvalue references and the special class functions (constructors, assignment operators, and destructor). We will demonstrate how to implement a move constructor and assignment operator using the following `Buffer` class:

```
class Buffer
{
  unsigned char* ptr;
  size_t length;
public:
  Buffer(): ptr(nullptr), length(0)
  {}

  explicit Buffer(size_t const size):
    ptr(new unsigned char[size] {0}), length(size)
  {}

  ~Buffer()
  {
    delete[] ptr;
  }

  Buffer(Buffer const& other):
    ptr(new unsigned char[other.length]),
  length(other.length)
  {
    std::copy(other.ptr, other.ptr + other.length, ptr);
  }

  Buffer& operator=(Buffer const& other)
  {
    if (this != &other)
    {
      delete[] ptr;

      ptr = new unsigned char[other.length];
      length = other.length;

      std::copy(other.ptr, other.ptr + other.length, ptr);
    }

    return *this;
  }

  size_t size() const { return length;}
  unsigned char* data() const { return ptr; }
};
```

How to do it…

To implement the move constructor for a class, do the following:

1. Write a constructor that takes an rvalue reference to the class type:

```
Buffer(Buffer&& other)
{
}
```

2. Assign all the data members from the rvalue reference to the current object. This can be done either in the body of the constructor, as follows, or in the initialization list, which is the preferred way:

```
ptr = other.ptr;
length = other.length;
```

3. Assign the data members from the rvalue reference to default values:

```
other.ptr = nullptr;
other.length = 0;
```

Put all together, the move constructor for the Buffer class looks like this:

```
Buffer(Buffer&& other):
{
  ptr = other.ptr;
  length = other.length;

  other.ptr = nullptr;
  other.length = 0;
}
```

To implement the move assignment operator for a class, do the following:

1. Write an assignment operator that takes an rvalue reference to the class type and returns a reference to it:

```
Buffer& operator=(Buffer&& other)
{
}
```

2. Check that the rvalue reference does not refer to the same object as `this`, and if they are different, perform steps 3 to 5:

```
if (this != &other)
{
}
```

3. Dispose all the resources (such as memory, handles, and so on) from the current object:

```
delete[] ptr;
```

4. Assign all the data members from the rvalue reference to the current object:

```
        ptr = other.ptr;
        length = other.length;
```

5. Assign the data members from the rvalue reference to default values:

```
        other.ptr = nullptr;
        other.length = 0;
```

6. Return a reference to the current object, regardless of whether steps 3 to 5 were executed or not:

```
        return *this;
```

Put all together, the move assignment operator for the `Buffer` class looks like this:

```
Buffer& operator=(Buffer&& other)
{
  if (this != &other)
  {
    delete[] ptr;

    ptr = other.ptr;
    length = other.length;

    other.ptr = nullptr;
    other.length = 0;
  }

  return *this;
}
```

How it works…

The move constructor and move assignment operator are provided by default by the compiler unless a user-defined copy constructor, move constructor, copy assignment operator, move assignment operator, or destructor exists already. When provided by the compiler, they perform a movement in a member-wise manner. The move constructor invokes the move constructors of the class data members recursively; similarly, the move assignment operator invokes the move assignment operators of the class data members recursively.

Move, in this case, represents a performance benefit for objects that are too large to copy (such as a string or container) or for objects that are not supposed to be copied (such as the `unique_ptr` smart pointer). Not all classes are supposed to implement both copy and move semantics. Some classes should only be movable, others both copyable and movable. On the other hand, it does not make much sense for a class to be copyable but not moveable, though this can be technically achieved.

Not all types benefit from move semantics. In the case of built-in types (such as `bool`, `int`, or `double`), arrays, or PODs, the move is actually a copy operation. On the other hand, move semantics provide a performance benefit in the context of rvalues, that is, temporary objects. An rvalue is an object that does not have a name; it lives temporarily during the evaluation of an expression and is destroyed at the next semicolon:

```
T a;
T b = a;
T c = a + b;
```

In the preceding example, `a`, `b`, and `c` are lvalues; they are objects that have a name which can be used to refer to the object at any point throughout its lifetime. On the other hand, when you evaluate the expression `a+b`, the compiler creates a temporary object (which, in this case, is assigned to `c`) and then destroyed (when a semicolon is encountered). These temporary objects are called rvalues because they usually appear on the right-hand side of an assignment expression. In C++11, we can refer to these objects through rvalue references, expressed with `&&`.

Move semantics are important in the context of rvalues. This is because they allow you to take ownership of the resources from the temporary object that is destroyed without the client being able to use it after the move operation is completed. On the other hand, lvalues cannot be moved; they can only be copied. This is because they can be accessed after the move operation, and the client expects the object to be in the same state. For instance, in the preceding example, the expression `b = a` assigns `a` to `b`. After this operation is complete, the object `a`, which is an lvalue, can still be used by the client and should be in the same state as it was before. On the other hand, the result of `a+b` is temporary and its data can be safely moved to `c`.

The move constructor is different than a copy constructor because it takes an rvalue reference to the class type `T(T&&)`, as opposed to an lvalue reference in the case of the copy constructor `T(T const&)`. Similarly, move assignment takes an rvalue reference, namely `T&`

`operator=(T&&)`, as opposed to an lvalue reference for the copy assignment operator, namely `T& operator=(T const &)`. This is true even though both return a reference to the `T&` class. The compiler selects the appropriate constructor or assignment operator based on the type of argument, rvalue, or lvalue.

When a move constructor/assignment operator exists, an rvalue is moved automatically. lvalues can also be moved, but this requires an explicit static cast to an rvalue reference. This can be done using the `std::move()` function, which basically performs a `static_cast<T&&>`:

```
std::vector<Buffer> c;
c.push_back(Buffer(100));  // move

Buffer b(200);
c.push_back(b);            // copy
c.push_back(std::move(b)); // move
```

After an object is moved, it must remain in a valid state. However, there is no requirement regarding what this state should be. For consistency, you should set all member fields to their default value (numerical types to 0, pointers to `nullptr`, booleans to `false`, and so on).

The following example shows the different ways in which `Buffer` objects can be constructed and assigned:

```
Buffer b1;                // default constructor
Buffer b2(100);           // explicit constructor
Buffer b3(b2);            // copy constructor
b1 = b3;                  // assignment operator
Buffer b4(std::move(b1)); // move constructor
b3 = std::move(b4);       // move assignment
```

There’s more…

As seen with the `Buffer` example, implementing both the move constructor and move assignment operator involves writing similar code (the entire code of the move constructor was also present in the move assignment operator). This can actually be avoided by calling the move assignment operator in the move constructor:

```
Buffer(Buffer&& other) : ptr(nullptr), length(0)
{
  *this = std::move(other);
}
```

There are two points that must be noticed in this example:

- Member initialization in the constructor’s initialization list is necessary because these members could potentially be used in the move assignment operator later on (such as the `ptr` member in this example).
- Static casting of `other` to an rvalue reference. Without this explicit conversion, the copy assignment operator would be called. This is because even if an rvalue is passed to this constructor as an argument, when it is assigned a name, it is bound to an lvalue. Therefore, `other` is actually an lvalue, and it must be converted into an rvalue reference in order to invoke the move assignment operator.

See also

- *Defaulted and deleted functions* recipe of Chapter 3, *Exploring Functions*

Implementing Patterns and Idioms

The recipes included in this chapter are as follows:

- Avoiding repetitive if…else statements in factory patterns
- Implementing the pimpl idiom
- Implementing the named parameter idiom
- Separating interfaces from implementations with the non-virtual interface idiom
- Handling friendship with the attorney-client idiom
- Static polymorphism with the curiously recurring template pattern
- Implementing a thread-safe singleton

Introduction

Design patterns are general reusable solutions that can be applied to common problems that appear in software development. Idioms are patterns, algorithms, or ways to structure the code in one or more programming languages. A great number of books have been written on design patterns. This chapter is not intended to reiterate them, but rather to show how to implement several useful patterns and idioms, with a focus on readability, performance, and robustness, in terms of modern C++.

Avoiding repetitive if…else statements in factory patterns

It is often the case that we end up writing repetitive `if...else` statements (or an equivalent `switch` statement) that do similar things, often with little variation and often done by copying and pasting with small changes. When the number of alternatives gets larger, the code becomes hard to both read and maintain. Repetitive `if...else` statements can be replaced with various techniques, such as polymorphism. In this recipe, we will see how to avoid `if...else` statements in factory patterns (a factory is a function or object that is used to create other objects) using a map of functions.

Getting ready

In this recipe, we will consider the following problem: building a system that can handle image files in various formats, such as bitmap, PNG, JPG, and so on. Obviously, the details are beyond the scope of this recipe; the part we are concerned with is creating objects that handle various image formats. For this, we will consider the following hierarchy of classes:

```
class Image {};
class BitmapImage : public Image {};
class PngImage : public Image {};
class JpgImage : public Image {};
```

On the other hand, we define an interface to a factory class that can create instances of the above classes, as well as a typical implementation using `if...else` statements:

```
struct IImageFactory
{
  virtual std::shared_ptr<Image> Create(std::string_view type) = 0;
};

struct ImageFactory : public IImageFactory
{
  virtual std::shared_ptr<Image>
  Create(std::string_view type) override
  {
    if (type == "bmp")
      return std::make_shared<BitmapImage>();
    else if (type == "png")
      return std::make_shared<PngImage>();
    else if (type == "jpg")
      return std::make_shared<JpgImage>();
    return nullptr;
  }
};
```

The goal of this recipe is to see how this implementation can be refactored to avoid the repetitive `if...else` statements.

How to do it…

Take the following steps to refactor the factory shown earlier to avoid using `if...else` statements:

1. Implement the factory interface:

```
struct ImageFactory : public IImageFactory
{
  virtual
  std::shared_ptr<Image> Create(std::string_view type) override
  {
    // continued with 2. and 3.
  }
};
```

2. Define a map where the key is the type of objects to create and the value is a function that creates objects:

```
static std::map<
  std::string,
  std::function<std::shared_ptr<Image>()>> mapping
{
  { "bmp", []() {return std::make_shared<BitmapImage>(); } },
  { "png", []() {return std::make_shared<PngImage>(); } },
  { "jpg", []() {return std::make_shared<JpgImage>(); } }
};
```

3. To create an object, look up the object type in the map and, if it is found, use the associated function to create a new instance of the type:

```
auto it = mapping.find(type.data());
if (it != mapping.end())
  return it->second();
return nullptr;
```

How it works…

The repetitive `if...else` statements in the first implementation are very similar — they check the value of the type parameter and create an instance of the appropriate `Image` class. If the argument to check was an integral type (for instance, an enumeration type), the sequence of `if...else` could have also be written in the form of a `switch` statement. That code can be used like this:

```
auto factory = ImageFactory{};
auto image = factory.Create("png");
```

Regardless of whether the implementation was using `if...else` statements or a `switch`, refactoring to avoid repetitive checks is relatively simple. In the refactored code, we used a map that has the key type `std::string` representing the type, that is, the name, of the image format, and the value is an `std::function<std::shared_ptr<Image>()>`. This is a wrapper for a function that takes no arguments and returns an `std::shared_ptr<Image>` (a `shared_ptr` of a derived class is implicitly converted to a `shared_ptr` of a base class).

Now that we have this map of functions that create objects, the actual implementation of the factory is much simpler; check the type of the object to be created in the map and, if present, use the associated value from the map as the actual function to create the object, or return `nullptr` if the object type is not present in the map.

This refactoring is transparent for the client code, as there are no changes in the way clients use the factory. On the other hand, this approach does require more memory to handle the static map, which, for some classes of applications, such as IoT, might be an important aspect. The example presented here is relatively simple, because the purpose is to demonstrate the concept. In real-world code, it might be necessary to create objects differently, such as using a different number of arguments and different types of arguments. However, this is not specific to the refactored implementation and the solution with `if...else`/`switch` statement needs to account for that too. Therefore, in practice, the solution to this problem that worked with `if...else` statements should also work with the map.

There’s more…

In the preceding implementation, the map is a local static to the virtual function, but it can also be a member of the class or even a global. The following implementation has the map defined as a static member of the class, and the objects are not created based on the format name, but on the type information, as returned by the `typeid` operator:

```
struct IImageFactoryByType
{
  virtual std::shared_ptr<Image> Create(std::type_info const & type)
  = 0;
};

struct ImageFactoryByType : public IImageFactoryByType
{
  virtual
  std::shared_ptr<Image> Create(std::type_info const & type)
  override
  {
    auto it = mapping.find(&type);
    if (it != mapping.end())
      return it->second();
    return nullptr;
  }
private:
  static std::map<
    std::type_info const *,
    std::function<std::shared_ptr<Image>()>> mapping;
};

std::map<
  std::type_info const *,
  std::function<std::shared_ptr<Image>()>> ImageFactoryByType::mapping
{
  {&typeid(BitmapImage),[](){return std::make_shared<BitmapImage>();}},
  {&typeid(PngImage),   [](){return std::make_shared<PngImage>();}},
  {&typeid(JpgImage),   [](){return std::make_shared<JpgImage>();}}
};
```

In this case, the client code is slightly different, because instead of passing a name representing the type to create, such as PNG, we pass the value returned by the `typeid` operator, such as `typeid(PngImage)`:

```
auto factory = ImageFactoryByType{};
auto movie = factory.Create(typeid(PngImage));
```

See also

- *Implementing the pimpl idiom*
- *Using shared_ptr to share a memory resource* recipe of Chapter 9, *Robustness and Performance*

Implementing the pimpl idiom

PIMPL stands for *pointer to implementation* (but is also known as the *Cheshire cat idiom* or the *compiler firewall idiom*) and is an opaque pointer technique that enables the separation of the implementation details from an interface. This has the advantage that it enables changing the implementation without modifying the interface and, therefore, avoiding the need to recompile the code that is using the interface. This has the potential of making libraries using the pimpl idiom on their ABIs that are backward compatible with older versions when only implementation details change. In this recipe, we will see how to implement the pimpl idiom using modern C++ features.

Getting ready

The reader is expected to be familiar with smart pointers and `std::string_view`, both discussed in previous chapters of this book.

To demonstrate the pimpl idiom in a practical manner, we will consider the following class that we will then refactor following the pimpl pattern. The class represents a control that has properties such as text, size, and visibility. Every time these properties are changed, the control is redrawn (in this mocked implementation, drawing means printing the value of the properties to the console):

```
class control
{
  std::string text;
  int width = 0;
  int height = 0;
  bool visible = true;

  void draw()
  {
    std::cout
      << "control " << std::endl
      << " visible: " << std::boolalpha << visible <<
         std::noboolalpha << std::endl
      << " size: " << width << ", " << height << std::endl
      << " text: " << text << std::endl;
  }
public:
  void set_text(std::string_view t)
  {
    text = t.data();
    draw();
  }

  void resize(int const w, int const h)
  {
    width = w;
    height = h;
    draw();
  }

  void show()
  {
    visible = true;
    draw();
  }

  void hide()
  {
    visible = false;
    draw();
  }
};
```

How to do it…

Take the following steps to implement the pimpl idiom, exemplified here by refactoring the `control` class shown earlier:

1. Put all private members, both data and functions, into a separate class. We will call this the *pimpl class* and the original class the *public class*.
2. In the header file of the public class, put a forward declaration to the pimpl class:

```
// in control.h
class control_pimpl;
```

3. In the public class definition, declare a pointer to the pimpl class using a `unique_ptr`. This should be the only private data member of the class:

```
class control
{
  std::unique_ptr<
    control_pimpl, void(*)(control_pimpl*)> pimpl;
public:
  control();
  void set_text(std::string_view text);
  void resize(int const w, int const h);
  void show();
  void hide();
};
```

4. Put the pimpl class definition in the source file of the public class. The pimpl class mirrors the public interface of the public class:

```
// in control.cpp
class control_pimpl
{
  std::string text;
  int width = 0;
  int height = 0;
  bool visible = true;

  void draw()
  {
     std::cout
       << "control " << std::endl
       << " visible: " << std::boolalpha << visible
       << std::noboolalpha << std::endl
       << " size: " << width << ", " << height << std::endl
       << " text: " << text << std::endl;
  }

public:
  void set_text(std::string_view t)
  {
    text = t.data();
    draw();
  }

  void resize(int const w, int const h)
  {
    width = w;
    height = h;
    draw();
  }

  void show()
  {
```

```
    visible = true;
    draw();
  }

  void hide()
  {
    visible = false;
    draw();
  }
};
```

5. The pimpl class is instantiated in the constructor of the public class:

```
control::control() :
  pimpl(new control_pimpl(),
        [](control_pimpl* pimpl) {delete pimpl; })
{}
```

6. Public class member functions call the corresponding member functions of the pimpl class:

```
void control::set_text(std::string_view text)
{
  pimpl->set_text(text);
}

void control::resize(int const w, int const h)
{
  pimpl->resize(w, h);
}

void control::show()
{
  pimpl->show();
}

void control::hide()
{
  pimpl->hide();
}
```

How it works…

The pimpl idiom the enables hiding the internal implementation of a class from the clients of the library or module the class is part of. This provides several benefits:

- A clean interface for a class that its clients see.
- Changes in the internal implementation do not affect the public interface, which enables binary backward compatibility for newer versions of a library (when the public interface remains unchanged).
- Clients of a class that uses this idiom do not need to be recompiled when changes to the internal implementation occur. This leads to lesser build times.
- The header file does not need to include the headers for the types and functions used in the private implementation. This again leads to lesser build times.

The benefits mentioned above do not come for free; there are also several drawbacks that need to be mentioned:

- There is more code to write and maintain.
- The code can arguably be less readable, as there is a level of indirection and all the implementation details need to be looked up in the other files. In this recipe, the the pimpl class definition was provided in the source file of the public class, but in practice, it could be in separate files.
- There is a slight runtime overhead because of the level of indirection from the public class to the pimpl class, but in practice, this is rarely significant.
- This approach does not work with protected members because these have to be available to the derived classes.
- This approach does not work with the private virtual functions, that have to appear in the class, either because they override functions from a base class, or have to be available for overriding in a derived class.

As a rule of thumb, when implementing the pimpl idiom, always put all private member data and functions, except for the virtual ones, in the pimpl class and leave the protected data members and functions and all private virtual functions in the public class.

In the example in this recipe, the `control_pimpl` class is basically identical to the original control class. In practice, where classes are larger, have virtual functions and protected members, and both functions and data, the pimpl class is not a complete equivalent of how the class would have looked like if it was not pimpled. Also, in practice, the pimpl class may require a pointer to the public class in order to call members that were not moved into the pimpl class.

Concerning the implementation of the refactored `control` class, the pointer to the `control_pimpl` object is managed by a `unique_ptr`. In the declaration of this pointer, we have used a custom deleter:

```
    std::unique_ptr<control_pimpl, void(*)(control_pimpl*)> pimpl;
```

The reason for this is that the control class has a destructor implicitly defined by the compiler, at a point where the `control_pimpl` type is still incomplete (that is, in the header). This would result in an error with the `unique_ptr` that cannot delete an incomplete type. The problem can be solved in two ways:

- Providing a user-defined destructor for the control class that is explicitly implemented (even if declared as `default`) after the complete definition of the `control_pimpl` class is available.
- Providing a custom deleter for the `unique_ptr`, as we did in this example.

There’s more…

The original `control` class was both copyable and movable:

```
control c;
c.resize(100, 20);
c.set_text("sample");
c.hide();

control c2 = c;             // copy
c2.show();

control c3 = std::move(c2); // move
c3.hide();
```

The refactored `control` class is only movable, not copyable. In order to make it both copyable and movable, we must provide the copy constructor and copy assignment operator and both the move constructor and move assignment operator. The latter ones can be defaulted, but the former ones must be explicitly implemented to create a new `control_pimpl` object from the object that it is copied from. The following code shows the implementation of the `control` class that is both copyable and movable:

```
class control_copyable
{
  std::unique_ptr<control_pimpl, void(*)(control_pimpl*)> pimpl;
public:
  control_copyable();
  control_copyable(control_copyable && op) noexcept;
  control_copyable& operator=(control_copyable && op) noexcept;
  control_copyable(const control_copyable& op);
  control_copyable& operator=(const control_copyable& op);

  void set_text(std::string_view text);
  void resize(int const w, int const h);
  void show();
  void hide();
};

control_copyable::control_copyable() :
  pimpl(new control_pimpl(),
        [](control_pimpl* pimpl) {delete pimpl; })
{}

control_copyable::control_copyable(control_copyable &&)
   noexcept = default;
control_copyable& control_copyable::operator=(control_copyable &&)
   noexcept = default;

control_copyable::control_copyable(const control_copyable& op)
   : pimpl(new control_pimpl(*op.pimpl),
           [](control_pimpl* pimpl) {delete pimpl; })
{}

control_copyable& control_copyable::operator=(
   const control_copyable& op)
{
  if (this != &op)
  {
    pimpl = std::unique_ptr<control_pimpl,void(*)(control_pimpl*)>(
               new control_pimpl(*op.pimpl),
               [](control_pimpl* pimpl) {delete pimpl; });
  }
  return *this;
}

// the other member functions
```

See also

- *Using unique_ptr to uniquely own a memory resource* recipe of Chapter 9, *Robustness and Performance*

Implementing the named parameter idiom

C++ supports only positional parameters, which means arguments are passed to a function based on the parameter's position. Other languages also support named parameters—that is, they specify parameter names when making a call and invoking arguments. This is particularly useful with parameters that have default values. A function may have parameters with default values, although they always appear after all the nondefaulted parameters. However, if you want to provide values for only some of the defaulted parameters, there is no way to do it without providing values for the arguments that are positioned before them in the function parameters list. A technique called the *named parameter idiom* provides a method to emulate named parameters, which we will explore in this recipe.

Getting ready

The `control` class represents a visual control, such as a button or an input and has properties such as numerical identifier, text, size, and visibility. These are provided to the constructor and, except for the ID, all the others have default values. In practice, such a class would have many more properties, such as text brush, background brush, border style, font size, font family, and many others. To exemplify the named parameter idiom, we will use the `control` class shown in the following code snippet.

```
class control
{
  int id_;
  std::string text_;
  int width_;
  int height_;
  bool visible_;
public:
  control(
    int const id,
    std::string_view text = "",
    int const width = 0,
    int const height = 0,
    bool const visible = false):
    id_(id), text_(text),
    width_(width), height_(height),
    visible_(visible)
  {}
};
```

How to do it…

To implement the named parameter idiom for a function (usually with many default parameters), do the following:

1. Create a class to wrap the parameters of the function:

```
class control_properties
{
  int id_;
  std::string text_;
  int width_;
  int height_;
  bool visible_;
};
```

2. The class or function that needs to access these properties could be declared as `friend` to avoid writing getters:

```
friend class control;
```

3. Every positional parameter of the original function that does not have a default value should become a positional parameter without a default value in the constructor of the class:

```
public:
  control_properties(int const id) :id_(id)
  {}
```

4. For every positional parameter of the original function that has a default value, there should be a function (with the same name) that sets the value internally and returns a reference to the class:

```
public:
  control_properties& text(std::string_view t)
  { text_ = t.data(); return *this; }

  control_properties& width(int const w)
  { width_ = w; return *this; }

  control_properties& height(int const h)
  { height_ = h; return *this; }

  control_properties& visible(bool const v)
  { visible_ = v; return *this; }
```

5. The original function should be modified, or an overload should be provided, to take an argument of the new class from which the property values would be read:

```
control(control_properties const & cp):
  id_(cp.id_),
  text_(cp.text_),
  width_(cp.width_),
  height_(cp.height_),
  visible_(cp.visible_)
{}
```

If we put all that together, the result is the following:

```
class control;

class control_properties
{
  int id_;
  std::string text_;
  int width_ = 0;
  int height_ = 0;
  bool visible_ = false;

  friend class control;
public:
  control_properties(int const id) :id_(id)
  {}

  control_properties& text(std::string_view t)
  { text_ = t.data(); return *this; }

  control_properties& width(int const w)
  { width_ = w; return *this; }

  control_properties& height(int const h)
  { height_ = h; return *this; }

  control_properties& visible(bool const v)
  { visible_ = v; return *this; }
};

class control
{
  int id_;
  std::string text_;
  int width_;
  int height_;
  bool visible_;
public:
  control(control_properties const & cp):
    id_(cp.id_),
    text_(cp.text_),
    width_(cp.width_),
    height_(cp.height_),
    visible_(cp.visible_)
  {}
};
```

How it works…

The initial `control` class had a constructor with many parameters. In real-world code, you can find examples like this where the number of parameters is much higher. A possible solution, often found in practice, is to group common Boolean type properties in bit flags, that could be passed together as a single integral argument (an example could be the border style for a control that defines the position where the border should be visible: top, bottom, left, right, or any combination of these four). Creating a control object with the initial implementation is done like this:

```
control c(1044, "sample", 100, 20, true);
```

The named parameter idiom has the advantage that it allows you to specify values only for the parameters that you want, in any order, using a name, which is much more intuitive than a fixed, positional order.

Although there isn't a single strategy for implementing the idiom, the example in this recipe is rather typical. The properties of the `control` class, provided as parameters in the constructor, have been put into a separate class, called `control_properties`, that declares the class `control` a friend class, to allow it to access its private data members without providing getters. This has the side effect that it limits the use of the `control_properties` outside the `control` class. The non-optional parameters of the constructor of the `control` are also non-optional parameters of the `control_properties` constructor. For all the other parameters with default values, the `control_properties` class defines a function with a relevant name that simply sets the data member to the provided argument and then returns a reference to `control_properties`. This enables the client to chain calls to these functions in any order.

The constructor of the control class has been replaced with a new one that has a single parameter, a constant reference to a `control_properties` object, whose data members are copied into the `control`'s data members. Creating a `control` object with the named parameter idiom implemented in this manner is done like in the following snippet:

```
control c(control_properties(1044)
          .visible(true)
          .height(20)
          .width(100));
```

See also

- *Separating interfaces and implementations with the non-virtual interface idiom*
- *Handling friendship with the attorney-client idiom*

Separating interfaces and implementations with the non-virtual interface idiom

Virtual functions provide customization points for a class, by allowing derived classes to modify implementations from a base class. When a derived class object is handled through a pointer or a reference to a base class, calls to overridden virtual functions end up invoking the overridden implementation from the derived class. On the over hand, a customization is an implementation detail, and a good design separates interfaces from implementation. The *non-virtual interface idiom*, proposed by Herb Sutter in an article about virtuality in *C/C++ Users Journal*, promotes the separation of concerns of interfaces and implementations by making (public) interfaces non-virtual and virtual functions private. Public virtual interfaces prevent a class from enforcing pre- and post-conditions on its interface. Users expecting an instance of a base class do not have a guarantee the expected behavior of a public virtual method is delivered, since it can be overridden in a derived class. This idiom helps enforcing the promised contract of an interface.

Getting ready

The reader should be familiar with aspects related to virtual functions, such as defining and overriding virtual functions, abstract classes, and pure specifiers.

How to do it…

Implementing this idiom requires following several simple design guidelines, formulated by Herb Sutter in the *C/C++ Users Journal*, 19(9), September 2001:

1. Make (public) interfaces non-virtual.
2. Make virtual functions private.
3. Make virtual functions protected only if the base implementation has to be called from a derived class.
4. Make the base class destructor either public and virtual or protected and nonvirtual.

The following example of a simple hierarchy of controls abides to all these four guidelines:

```
class control
{
private:
  virtual void paint() = 0;
protected:
  virtual void erase_background()
  {
    std::cout << "erasing control background..." << std::endl;
  }
public:
  void draw()
  {
    erase_background();
    paint();
  }

  virtual ~control() {}
};

class button : public control
{
private:
  virtual void paint() override
  {
    std::cout << "painting button..." << std::endl;
  }
protected:
  virtual void erase_background() override
  {
    control::erase_background();
    std::cout << "erasing button background..." << std::endl;
  }
};

class checkbox : public button
{
private:
  virtual void paint() override
  {
    std::cout << "painting checkbox..." << std::endl;
  }
protected:
  virtual void erase_background() override
  {
    button::erase_background();
    std::cout << "erasing checkbox background..." << std::endl;
  }
};
```

How it works…

The NVI idiom uses the *template method* design pattern that allows derived classes to customize parts (that is, steps) of a base class functionality (that is, algorithm). This is done by splitting the overall algorithm into smaller parts, each of them implemented by a virtual function. The base class may provide, or not, a default implementation, and the derived classes could override them while maintaining the overall structure and meaning of the algorithm.

The core principles of the NVI idiom is that virtual functions should not be public; they should be either private or protected, in case the base class implementation could be called from a derived class. The interface of a class, the public part accessible to its clients, should comprise exclusively of nonvirtual functions. This provides several advantages:

- It separates the interface from the details of implementation that are no longer exposed to the client.
- It enables the changing of the details of the implementation without altering the public interface and without requiring changes to the client code, therefore, making base classes more robust.
- It allows a class to have sole control of its interface. If the public interface contains virtual methods, a derived class can alter the promised functionality, and therefore, the class cannot ensure its preconditions and postconditions. When all virtual methods (except for the destructor) are not accessible to its clients, the class can enforce pre- and post-conditions on its interface.

A special mention of the destructor of a class is required for this idiom. It is often stressed that base class destructors should be virtual so that objects can be deleted polymorphically (through a pointer or references to a base class). Destructing objects polymorphically when the destructor is not virtual incurs undefined behavior. However, not all base classes are intended to be deleted polymorphically. For those particular cases, the base class destructor should not be virtual. However, it should also not be public, but protected.

The example from the previous section defines a hierarchy of classes representing visual controls:

- `control` is the base class, but there are derived classes, such as `button` and `checkbox` that are a type of button and, therefore, are derived from this class.
- The only functionality defined by the `control` class is drawing the controls. The `draw()` method is nonvirtual, but it calls two virtual methods, `erase_background()` and `paint()`, to implement the two phases of drawing the control.
- `erase_background()` is a protected virtual method because derived classes need to call it in their own implementation.
- `paint()` is a private pure virtual method. Derived classes must implement it, but are not

supposed to call a base implementation.
- The destructor of the class control is public and virtual because objects are expected to be deleted polymorphically.

An example of using these classes is shown as follows. Instances of these classes are managed by smart pointers to the base class:

```
    std::vector<std::shared_ptr<control>> controls;

    controls.push_back(std::make_shared<button>());
    controls.push_back(std::make_shared<checkbox>());

    for (auto& c : controls)
      c->draw();
```

The output of this program is the following:

```
erasing control background...
erasing button background...
painting button...
erasing control background...
erasing button background...
erasing checkbox background...
painting checkbox...
destroying button...
destroying control...
destroying checkbox...
destroying button...
destroying control...
```

The NVI idiom introduces a level of indirection, when a public function calls a non-public virtual function that does the actual implementation. In the previous example, the `draw()` method called several other functions, but in many cases it could be only one call:

```
    class control
    {
    protected:
      virtual void initialize_impl()
      {
        std::cout << "initializing control..." << std::endl;
      }
    public:
      void initialize()
      {
        initialize_impl();
      }
    };

    class button : public control
    {
    protected:
      virtual void initialize_impl()
      {
        control::initialize_impl();
        std::cout << "initializing button..." << std::endl;
      }
    };
```

In this example, the class control has an additional method called `initialize()` (the previous content of the class was not shown to keep it simple) that calls a single non-public virtual method called `initialize_impl()`, implemented differently in each derived class. This does not incur much overhead—if any at all—since simple functions like this are most likely inlined by the compiler anyway.

See also

- *Use override and final for virtual methods* recipe of Chapter 1, *Learning Modern Core Language Features*

Handling friendship with the attorney-client idiom

Granting functions and classes access to the non-public parts of a class with a friend declaration has been usually seen as a sign of bad design, as friendship breaks encapsulation and couples classes and functions. Friends, whether they are classes or functions, get access to all the private parts of a class, although they may only need to access parts of it. The attorney-client idiom provides a simple mechanism to restrict friends access to only designated private parts of a class.

Getting ready

You must be familiar with how friendship is declared and how it works.

To demonstrate how to implement this idiom, we will consider the following classes: `Client`, which has some private member data and functions (the public interface is not important here) and `Friend`, which is supposed to access only parts of the private details, for instance, `data1` and `action1()`, but has access to everything:

```
class Client
{
  int data_1;
  int data_2;

  void action1() {}
  void action2() {}

  friend class Friend;
public:
  // public interface
};

class Friend
{
public:
  void access_client_data(Client& c)
  {
    c.action1();
    c.action2();
    auto d1 = c.data_1;
    auto d2 = c.data_1;
  }
};
```

How to do it…

Take the following steps to restrict a friend's access to the private parts of a class:

1. In the client class that provides access to its private parts to a friend, declare the friendships to an intermediate class, called the `Attorney` class:

```
class Client
{
  int data_1;
  int data_2;

  void action1() {}
  void action2() {}

  friend class Attorney;
public:
  // public interface
};
```

2. Create a class that contains only private (inline) functions that access the private parts of the client. This intermediate class allows the actual friend to access its private parts:

```
class Attorney
{
  static inline void run_action1(Client& c)
  {
    c.action1();
  }

  static inline int get_data1(Client& c)
  {
    return c.data_1;
  }

  friend class Friend;
};
```

3. In the `Friend` class, access the private parts of only the `Client` class indirectly through the `Attorney` class:

```
class Friend
{
public:
  void access_client_data(Client& c)
  {
    Attorney::run_action1(c);
    auto d1 = Attorney::get_data1(c);
  }
};
```

How it works…

The attorney-client idiom lays out a simple mechanism to restrict access to the private parts of the client by introducing a middleman, the attorney. Instead of providing friendship directly to those using its internal state, the client class offers friendship to an attorney, which in turn provides access to a restricted set of private data or functions of the client. It does so by defining private static functions. Usually, these are also inline functions, which avoids any runtime overhead due to the level of indirection the attorney class introduces. The client's friend gets access to its private parts by actually using the private parts of the attorney. This idiom is called *attorney-client* because it is similar to the way an attorney-client relationship works, with the attorney knowing all the secrets of the client, but exposing only some of them to other parties.

In practice, it might be necessary to create more than one attorney for a client class if different friend classes or functions must access different private parts.

On the other hand, friendship is not inheritable, which means that a class or function that is friend to a class `B` is not friend with a class `D` that is derived from `B`. However, virtual functions overridden in `D` are still accessible polymorphically through a pointer or reference to `B` from a friend class. Such an example is shown as follows; calling the `run()` method from `F` prints `base` and `derived`:

```
class B
{
  virtual void execute() { std::cout << "base" << std::endl; }
  friend class BAttorney;
};

class D : public B
{
  virtual void execute() override
  { std::cout << "derived" << std::endl; }
};

class BAttorney
{
  static inline void execute(B& b)
  {
    b.execute();
  }
  friend class F;
};

class F
{
public:
  void run()
  {
    B b;
    BAttorney::execute(b); // prints 'base'

    D d;
    BAttorney::execute(d); // prints 'derived'
  }
};

F f;
f.run();
```

There’s more

There are always trade-offs for using a design pattern, and this one is not an exception. There are situations when using this pattern may lead to a too much overhead on development, testing, and maintenance. However, the pattern could prove extremely valuable for some types of applications, such as extensible frameworks.

See also

- *Implementing the pimpl idiom*

Static polymorphism with the curiously recurring template pattern

Polymorphism is the ability to have multiple forms for the same interface. Virtual functions allow derived classes to override implementations from a base class. They represent the most common elements of a form of polymorphism called *runtime polymorphism* because the decision to call a particular virtual function from the class hierarchy happens at runtime. It is also called *late binding*, because the binding between a function call and the invocation of the function happens late, during the execution of the program. The opposite of this is called *early binding*, *static polymorphism*, or *compile time polymorphism* because it occurs at compile time through functions and operators overloading. On the other hand, a technique called the *curiously recurring template pattern* (or *CRTP*) allows simulating the virtual functions-based runtime polymorphism at compile time, by deriving classes from a base class template parameterized with the derived class. This technique is used extensively in some libraries, including the *Microsoft's Active Template Library* (*ATL*) and *Windows Template Library* (*WTL*).

Getting ready

To demonstrate how the CRTP works, we will revisit the example with the hierarchy of control classes implemented in the *Separating interfaces from implementations with the non-virtual interface idiom* recipe. We will define a set of control classes that have functionalities such as drawing the control, that is, (in our example) an operation done in two phases: erasing the background and then painting the control.

How to do it…

To implement the curiously recurring template pattern in order to achieve static polymorphism, do the following:

1. Provide a class template that will represent the base class for other classes that should be treated polymorphically at compile time. Polymorphic functions are invoked from this class:

```
template <class T>
class control
{
public:
  void draw()
  {
    static_cast<T*>(this)->erase_background();
    static_cast<T*>(this)->paint();
  }
};
```

2. Derived classes use the class template as their base class; the derived class is also the template argument for the base class. The derived class implements the functions that are invoked from the base class:

```
class button : public control<button>
{
public:
  void erase_background()
  {
    std::cout << "erasing button background..." << std::endl;
  }

  void paint()
  {
    std::cout << "painting button..." << std::endl;
  }
};

class checkbox : public control<checkbox>
{
public:
  void erase_background()
  {
    std::cout << "erasing checkbox background..."
              << std::endl;
  }

  void paint()
  {
    std::cout << "painting checkbox..." << std::endl;
  }
};
```

3. Function templates can handle derived classes polymorphically through a pointer or reference to the base class template:

```
template <class T>
void draw_control(control<T>& c)
{
  c.draw();
}

button b;
```

```
        draw_control(b);

        checkbox c;
        draw_control(c);
```

How it works…

Virtual functions can represent a performance issue, especially when they are small and called multiple times in a loop. Modern hardware has made most of these situations rather irrelevant, but there are still some categories of applications where performance is critical and any performance gains are important. The curiously recurring template pattern enables the simulation of virtual calls at compile time using metaprogramming that eventually translates to functions overloading.

This pattern may look rather strange at a first glance, but it is perfectly legal. The idea is to derive a class from a base class that is a template class and to pass the derived class itself for the type template parameter of the base class. The base class then makes calls to the derived class functions. In our example, `control<button>::draw()` is declared before the `button` class is known to the compiler. However, the `control` class is a class template, that means, it is instantiated only when the compiler encounters code that uses it. At that point, the `button` class, in this example, is already defined and known to the compiler, so calls to `button::erase_background()` and `button::paint()` can be made.

To invoke the functions from the derived class, we must first obtain a pointer to the derived class. That is done with a `static_cast` conversion, as seen in `static_cast<T*>(this)->erase_background()`. If this has to be done many times, the code can be simplified by providing a private function to do that:

```
template <class T>
class control
{
  T* derived() { return static_cast<T*>(this); }
public:
  void draw()
  {
    derived()->erase_background();
    derived()->paint();
  }
};
```

There are some pitfalls when using the CRTP that you must be aware of:

- All the functions in the derived classes that are called from the base class template must be public; otherwise, the base class specialization must be declared a friend of the derived class:

```
class button : public control<button>
{
private:
  friend class control<button>;
  void erase_background()
  {
    std::cout << "erasing button background..." << std::endl;
  }

  void paint()
  {
    std::cout << "painting button..." << std::endl;
  }
};
```

- It is not possible to store in a homogeneous container, such as a `vector` or `list`, objects of CRTP types because each base class is a unique type (such as `control<button>` and `control<checkbox>`). If this is actually necessary, then a workaround can be used to implement it. This will be discussed and exemplified in the next section.
- When using this technique, the size of a program may increase, because of the way templates are instantiated.

There’s more…

When objects of types implementing the CRTP need to be stored homogeneously in a container, an additional idiom must be used. The base class template must be itself derived from another class with pure virtual functions (and a virtual public destructor). To exemplify this on the `control` class, the following changes are necessary:

```
class controlbase
{
public:
  virtual void draw() = 0;
  virtual ~controlbase() {}
};

template <class T>
class control : public controlbase
{
public:
  virtual void draw() override
  {
    static_cast<T*>(this)->erase_background();
    static_cast<T*>(this)->paint();
  }
};
```

There are no changes required to the derived classes, such as `button` and `checkbox`. Then, we can store pointers to the abstract class in a container, such as `std::vector`, as shown as follows:

```
void draw_controls(std::vector<std::shared_ptr<controlbase>>& v)
{
  for (auto & c : v)
  {
    c->draw();
  }
}

std::vector<std::shared_ptr<controlbase>> v;
v.emplace_back(std::make_shared<button>());
v.emplace_back(std::make_shared<checkbox>());

draw_controls(v);
```

See also

- *Implementing the pimpl idiom*
- *Separating interfaces from implementations with the non-virtual interface idiom*

Implementing a thread-safe singleton

Singleton is probably one of the most well-known design patterns. It restricts the instantiation of a single object of a class, something that is necessary in some cases, although many times the use of a singleton is rather an anti-pattern that can be avoided with other design choices. Since a singleton means a single instance of a class is available to an entire program, it is likely that such a unique instance might be accessible from different threads. Therefore, when you implement a singleton, you should also make it thread-safe. Before C++11, doing that was not an easy job, and a double-checked locking technique was the typical approach. However, Scott Meyers and Andrei Alexandrescu showed, in a paper called *C++ and the Perils of Double-Checked Locking*, that using this pattern did not guarantee a thread-safe singleton implementation in portable C++. Fortunately, this changed in C++11, and this recipe shows how to write one in modern C++.

Getting ready

For this recipe, you need to know how static storage duration and internal linkage and deleted and defaulted functions work. You should also first read the previous recipe, *Static polymorphism with the curiously recurring template pattern*, if you have not done that yet and are not familiar with that pattern because we will use it later in this recipe.

How to do it…

To implement a thread-safe singleton, you should do the following:

1. Define the singleton class:

```
class Singleton
{
};
```

2. Make the default constructor private:

```
private:
  Singleton() {}
```

3. Make the copy constructor and copy assignment operator `public` and `delete`:

```
public:
  Singleton(Singleton const &) = delete;
  Singleton& operator=(Singleton const&) = delete;
```

4. The function that creates and returns the single instance should be static and should return a reference to the class type. It should declare a static object of the class type and return a reference to it:

```
public:
  static Singleton& instance()
  {
    static Singleton single;
    return single;
  }
```

How it works…

Since singleton objects are not supposed to be created by the user directly, all constructors are either private or public and `deleted`. The default constructor is private and not deleted because an instance of the class must be actually created in the class code. A static function, called `instance()`, in this implementation, returns the single instance of the class.

Though most implementations return a pointer, it actually makes more sense to return a reference, as there is no circumstance under which this function would return a null pointer (no object).

The implementation of the `instance()` method may look simplistic and not thread-safe at a first glance, especially if you are familiar with the *double-checked locking pattern* (*DCLP*). In C++11, this is actually no longer necessary due to a key detail of how objects with static storage durations are initialized. Initialization happens only once, even if several threads attempt to initialize the same static object at the same time. The responsibility of the DCLP has been moved from the user to the compiler, although the compiler may use another technique to guarantee the result.

The following quote from the C++ standard, paragraph 6.7.4, defines the rules for static objects initialization (the highlight is the part related to concurrent initialization):

The zero-initialization (8.5) of all block-scope variables with static storage duration (3.7.1) or thread storage duration (3.7.2) is performed before any other initialization takes place. Constant initialization (3.6.2) of a block-scope entity with static storage duration, if applicable, is performed before its block is first entered. An implementation is permitted to perform early initialization of other block-scope variables with static or thread storage duration under the same conditions that an implementation is permitted to statically initialize a variable with static or thread storage duration in namespace scope (3.6.2). Otherwise, such a variable is initialized the first time control passes through its declaration; such a variable is considered initialized upon the completion of its initialization. If the initialization exits by throwing an exception, the initialization is not complete, so it will be tried again the next time control enters the declaration. ***If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization.*** *If control re-enters the declaration recursively while the variable is being initialized, the behavior is undefined.*

The static local object has storage duration, but it is instantiated only when it is first used (at the first call to the method `instance()`). The object is deallocated when the program exists. As a side note, the only possible advantage of returning a pointer and not a reference is the ability to delete this single instance at some point, before the program exists, and then maybe recreate it. This again does not make too much sense, as it conflicts with the idea of a single, global instance of a class, accessible at any point from any place in the program.

There’s more…

There might be situations in larger code bases where you need more than one singleton type. In order to avoid writing the same pattern several times, you can implement it in a generic way. For this, we need to employ the *curiously recurring template pattern* (or *CRTP*) seen in the previous recipe. The actual singleton is implemented as a class template. The `instance()` method creates and returns an object of the type template parameter, which will be the derived class:

```
template <class T>
class SingletonBase
{
protected:
  SingletonBase() {}
public:
  SingletonBase(SingletonBase const &) = delete;
  SingletonBase& operator=(SingletonBase const&) = delete;

  static T& instance()
  {
    static T single;
    return single;
  }
};

class Single : public SingletonBase<Single>
{
  Single() {}
  friend class SingletonBase<Single>;
public:
  void demo() { std::cout << "demo" << std::endl; }
};
```

The `Singleton` class from the previous section has become the `SingletonBase` class template. The default constructor is no longer private but protected because it must be accessible from the derived class. In this example, the class that needs to have a single object instantiated is called `Single`. Its constructors must be private, but the default constructor must also be available to the base class template; therefore, `SingletonBase<Single>` is a friend of the `Single` class.

See also

- *Static polymorphism with the curiously recurring template pattern*
- *Defaulted and deleted functions* recipe of Chapter 3, *Exploring Functions*

Exploring Testing Frameworks

This chapter includes the following recipes:

- Getting started with Boost.Test
- Writing and invoking tests with Boost.Test
- Asserting with Boost.Test
- Using test fixtures with Boost.Test
- Controlling output with Boost.Test
- Getting started with Google Test
- Writing and invoking tests with Google Test
- Asserting with Google Test
- Using test fixtures with Google Test
- Controlling output with Google Test
- Getting started with Catch
- Writing and invoking tests with Catch
- Asserting with Catch
- Controlling output with Catch

Introduction

Testing the code is an important part of software development. Although there is no support for testing in the C++ standard, there is a large variety of frameworks for unit testing C++ code. The purpose of this chapter is to get you started with several modern and widely used testing frameworks that enable you to write portable testing code. The frameworks discussed in this chapter were chosen due to their rich capabilities, the ease with which they can be used to write and execute tests, extensibility, and customization.

Getting started with Boost.Test

Boost.Test is one of the oldest and most popular C++ testing frameworks. It provides an easy-to-use set of APIs for writing tests and organizing them into test cases and test suites. It has good support for asserting, exception handling, fixtures, and other important features required for a testing framework. Throughout the next few recipes, we will explore the most important features it has which enable you to write unit tests. In this recipe, we will see how to install the framework and create a simple test project.

Getting ready

The Boost.Test framework has a macro-based API. Although you only need to use the supplied macros for writing tests, a good understanding of macros is recommended if you want to use the framework well.

How to do it…

In order to set up your environment to use Boost.Test, do the following:

1. Download the latest version of the Boost library from http://www.boost.org/.
2. Unzip the content of the archive.
3. Build the library using the provided tools and scripts in order to use either the static or shared library variant. This step is not necessary if you only plan to use the header-only version of the library.

To create your first test program using the header-only variant of the Boost.Test library, do the following:

1. Create a new empty C++ project.
2. Do the necessary setup specific to the development environment you are using to make the boost `main` folder available to the project for including header files.
3. Add a new source file to the project with the following content:

```
#define BOOST_TEST_MODULE My first test module
#include <boost/test/included/unit_test.hpp>

BOOST_AUTO_TEST_CASE(first_test_function)
{
  BOOST_TEST(true);
}
```

4. Build and run the project.

How it works…

The library can be downloaded along with other Boost libraries. In this book, I used version 1.63, but the features discussed in these recipes will probably be available for many future versions. The `Test` library comes in three variants:

- **Single header**: This enables you to write test programs without building the library; you just need to include a single header. Its limitation is that you can only have a single translation unit for the module; however, you can still split the module into multiple header files so that you can separate different test suites in different files.
- **Static library**: This enables you to split a module across different translation units, but the library needs to be built first as a static library.
- **Shared library**: This enables the same scenario as that of the static library. However, it has the advantage that, for programs with many test modules, this library is linked only once and not once for each module, resulting in a smaller binary size. However, in this case, the shared library must be available at runtime.

For simplicity, we will use the single-header variant in this book. In the case of static and shared library variants, you'd need to build the library. The downloaded archive contains scripts for building the library. However, the exact steps vary depending on the platform and the compiler; they will not be covered here but are available online.

There are several terms and concepts that you need to understand in order to use the library:

- **Test module** is a program that performs tests. There are two types of modules: **single-file** (when you use the single-header variant) and **multifile** (when you use either the static or shared variant).
- **Test assertion** is a condition that is checked by a test module.
- **Test case** is a group of one or more test assertions that is independently executed and monitored by a test module so that, if it fails or leaks uncaught exceptions, the execution of other tests would not be stopped.
- **Test suite** is a collection of one or more test cases or test suites.
- **Test unit** is either a test case or test suite.
- **Test tree** is a hierarchical structure of test units. In this structure, test cases are leaves and test suites are non-leaves.
- **Test runner** is a component that, given a test tree, performs the necessary initialization, execution of tests, and results reporting.
- **Test report** is the report produced by the test runner from the execution of the tests.
- **Test log** is the recording of all the events that occur during the execution of the test module.
- **Test setup** is the part of the test module responsible for the initialization of the framework, construction of the test tree, and individual test case setups.
- **Test cleanup** is a part of the test module responsible for cleanup operations.

- **Test fixture** is a pair of setup and cleanup operations that are invoked for multiple test units in order to avoid repetitive code.

With these concepts defined, it is possible to explain the sample code listed earlier:

1. `#define BOOST_TEST_MODULE My first test module` defines a stub for module initialization and sets a name for the main test suite. This must be defined before you include any library header.
2. `#include <boost/test/included/unit_test.hpp>` includes the single-header library, which includes all the other necessary headers.
3. `BOOST_AUTO_TEST_CASE(first_test_function)` declares a test case without parameters (`first_test_function`) and automatically registers it to be included in the test tree as part of the enclosing test suite. In this example, the test suite is the main test suite defined by `BOOST_TEST_MODULE`.
4. `BOOST_TEST(true);` performs a test assertion.

The output of executing this test module is as follows:

```
Running 1 test case...
*** No errors detected
```

There’s more…

If you don’t want the library to generate the `main()` function but want to write it yourself, then you need to define a couple more macros—`BOOST_TEST_NO_MAIN` and `BOOST_TEST_ALTERNATIVE_INIT_API`—before you include any of the library headers. Then, in the `main()` function that you supply, invoke the default test runner called `unit_test_main()` by providing the default initialization function called `init_unit_test()` as an argument, as shown in the following code snippet:

```
#define BOOST_TEST_MODULE My first test module
#define BOOST_TEST_NO_MAIN
#define BOOST_TEST_ALTERNATIVE_INIT_API
#include <boost/test/included/unit_test.hpp>

BOOST_AUTO_TEST_CASE(first_test_function)
{
  BOOST_TEST(true);
}

int main(int argc, char* argv[])
{
  return boost::unit_test::unit_test_main(init_unit_test, argc, argv);
}
```

It is also possible to customize the initialization function of the test runner. In this case, you must remove the definition of the `BOOST_TEST_MODULE` macro and instead write an initialization function that takes no arguments and returns a `bool` value:

```
#define BOOST_TEST_NO_MAIN
#define BOOST_TEST_ALTERNATIVE_INIT_API
#include <boost/test/included/unit_test.hpp>

BOOST_AUTO_TEST_CASE(first_test_function)
{
  BOOST_TEST(true);
}

bool custom_init_unit_test()
{
  std::cout << "test runner custom init" << std::endl;
  return true;
}

int main(int argc, char* argv[])
{
  return boost::unit_test::unit_test_main(
    custom_init_unit_test, argc, argv);
}
```

It is possible to customize the initialization function without writing the `main()` function yourself. In this case, the `BOOST_TEST_NO_MAIN` macro should not be defined and the initialization function should be called `init_unit_test()`.

See also

- *Writing and invoking tests with Boost.Test*

Writing and invoking tests with Boost.Test

The library provides both an automatic and manual way of registering test cases and test suites to be executed by the test runner. Automatic registration is the simplest way because it enables you to construct a test tree just by declaring test units. In this recipe, we will see how to create test suites and test cases, using the single-header version of the library, and how to run tests.

Getting ready

To exemplify the creation of test suites and test cases, we will use the following class which represents a three-dimensional point:

```
class point3d
{
  int x_;
  int y_;
  int z_;
public:
  point3d(int const x = 0,
          int const y = 0,
          int const z = 0):x_(x), y_(y), z_(z) {}

  int x() const { return x_; }
  point3d& x(int const x) { x_ = x; return *this; }
  int y() const { return y_; }
  point3d& y(int const y) { y_ = y; return *this; }
  int z() const { return z_; }
  point3d& z(int const z) { z_ = z; return *this; }

  bool operator==(point3d const & pt) const
  {
    return x_ == pt.x_ && y_ == pt.y_ && z_ == pt.z_;
  }

  bool operator!=(point3d const & pt) const
  {
    return !(*this == pt);
  }

  bool operator<(point3d const & pt) const
  {
    return x_ < pt.x_ || y_ < pt.y_ || z_ < pt.z_;
  }

  friend std::ostream& operator<<(std::ostream& stream,
                                  point3d const & pt)
  {
    stream << "(" << pt.x_ << "," << pt.y_ << "," << pt.z_ << ")";
    return stream;
  }

  void offset(int const offsetx, int const offsety, int const offsetz)
  {
    x_ += offsetx;
    y_ += offsety;
    z_ += offsetz;
  }

  static point3d origin() { return point3d{}; }
};
```

Before you go further, notice that the test cases in this recipe contain erroneous tests on purpose, so that they would produce failures.

How to do it…

Use the following macros to create test units:

- To create a test suite, use `BOOST_AUTO_TEST_SUITE(name)` and `BOOST_AUTO_TEST_SUITE_END()`:

```
BOOST_AUTO_TEST_SUITE(test_construction)
// test cases
BOOST_AUTO_TEST_SUITE_END()
```

- To create a test case, use `BOOST_AUTO_TEST_CASE(name)`. Test cases are defined between `BOOST_AUTO_TEST_SUITE(name)` and `BOOST_AUTO_TEST_SUITE_END()` , as shown in the following code snippet:

```
BOOST_AUTO_TEST_CASE(test_constructor)
{
  auto p = point3d{ 1,2,3 };
  BOOST_TEST(p.x() == 1);
  BOOST_TEST(p.y() == 2);
  BOOST_TEST(p.z() == 4); // will fail
}

BOOST_AUTO_TEST_CASE(test_origin)
{
  auto p = point3d::origin();
  BOOST_TEST(p.x() == 0);
  BOOST_TEST(p.y() == 0);
  BOOST_TEST(p.z() == 0);
}
```

- To create a nested test suite, define a test suite inside another test suite:

```
BOOST_AUTO_TEST_SUITE(test_operations)
BOOST_AUTO_TEST_SUITE(test_methods)

BOOST_AUTO_TEST_CASE(test_offset)
{
  auto p = point3d{ 1,2,3 };
  p.offset(1, 1, 1);
  BOOST_TEST(p.x() == 2);
  BOOST_TEST(p.y() == 3);
  BOOST_TEST(p.z() == 3); // will fail
}

BOOST_AUTO_TEST_SUITE_END()
BOOST_AUTO_TEST_SUITE_END()
```

- To add decorators to a test unit, add an additional parameter to the test unit's macros. Decorators could include description, label, precondition, dependency, fixture, and so on. Refer to the following code snippet which illustrates this:

```
BOOST_AUTO_TEST_SUITE(test_operations)
BOOST_AUTO_TEST_SUITE(test_operators)

BOOST_AUTO_TEST_CASE(
  test_equal,
  *boost::unit_test::description("test operator==")
  *boost::unit_test::label("opeq"))
{
  auto p1 = point3d{ 1,2,3 };
  auto p2 = point3d{ 1,2,3 };
  auto p3 = point3d{ 3,2,1 };
  BOOST_TEST(p1 == p2);
```

```
  BOOST_TEST(p1 == p3); // will fail
}

BOOST_AUTO_TEST_CASE(
  test_not_equal,
  *boost::unit_test::description("test operator!=")
  *boost::unit_test::label("opeq")
  *boost::unit_test::depends_on(
    "test_operations/test_operators/test_equal"))
{
  auto p1 = point3d{ 1,2,3 };
  auto p2 = point3d{ 3,2,1 };
  BOOST_TEST(p1 != p2);
}

BOOST_AUTO_TEST_CASE(test_less)
{
  auto p1 = point3d{ 1,2,3 };
  auto p2 = point3d{ 1,2,3 };
  auto p3 = point3d{ 3,2,1 };
  BOOST_TEST(!(p1 < p2));
  BOOST_TEST(p1 < p3);
}

BOOST_AUTO_TEST_SUITE_END()
BOOST_AUTO_TEST_SUITE_END()
```

To execute the tests, do the following:

- To execute the entire test tree, run the program (the test module) without any parameters:

```
chapter11bt_02.exe

Running 6 test cases...
f:/chapter11bt_02/main.cpp(12): error: in "test_construction/test_
constructor": check p.z() == 4 has failed [3 != 4]
f:/chapter11bt_02/main.cpp(35): error: in "test_operations/test_
methods/test_offset": check p.z() == 3 has failed [4 != 3]
f:/chapter11bt_02/main.cpp(55): error: in "test_operations/test_
operators/test_equal": check p1 == p3 has failed [(1,2,3) !=
(3,2,1)]
*** 3 failures are detected in the test module "Testing point 3d"
```

- To execute a single test suite, run the program with the argument `run_test` specifying the path of the test suite:

```
chapter11bt_02.exe --run_test=test_construction

Running 2 test cases...
f:/chapter11bt_02/main.cpp(12): error: in "test_construction/test_
constructor": check p.z() == 4 has failed [3 != 4]
*** 1 failure is detected in the test module "Testing point 3d"
```

- To execute a single test case, run the program with the argument `run_test` specifying the path of the test case:

```
chapter11bt_02.exe --run_test=test_construction/test_origin

Running 1 test case...
*** No errors detected
```

- To execute a collection of test suites and test cases defined under the same label, run the program with the argument `run_test` specifying the label name prefixed with `@`:

```
chapter11bt_02.exe --run_test=@opeq
```

```
Running 2 test cases...
f:/chapter11bt_02/main.cpp(56): error: in "test_operations/test_
operators/test_equal": check p1 == p3 has failed [(1,2,3) !=
(3,2,1)]
*** 1 failure is detected in the test module "Testing point 3d"
```

How it works…

A test tree is constructed from test suites and test cases. A test suite can contain one or more test cases and other nested test suites as well. Test suites are similar to namespaces in the sense that they can be stopped and restarted multiple times in the same file or in different files. Automatic registration of test suites is done with the macros `BOOST_AUTO_TEST_SUITE`, which requires a name, and `BOOST_AUTO_TEST_SUITE_END`. Automatic registration of test cases is done with `BOOST_AUTO_TEST_CASE`. Test units (whether cases or suites) become members of the closest test suite. Test units defined at the file scope level become members of the master test suite—the implicit test suite created with the `BOOST_TEST_MODULE` declaration.

Both test suites and test cases can be decorated with a series of attributes that affect how test units would be processed during the execution of the test module. The currently supported decorators are as follows:

- `depends_on`: This indicates a dependency between the current test unit and a designated test unit.
- `description`: This provides a semantic description of a test unit.
- `enabled` / `disabled`: These set the default run status of a test unit to either `true` or `false`.
- `enable_if`: This sets the default run status of a test unit to either `true` or `false`, depending on the evaluation of a compile-time expression.
- `fixture`: This specifies a pair of functions (startup and cleanup) to be called before and after the execution of a test unit.
- `label`: With this, you can associate a test unit with a label. The same label can be used for multiple test units, and a test unit can have multiple labels.
- `precondition`: This associates a predicate with a test unit, which is used at runtime to determine the run status of the test unit.

If the execution of a test case results in an unhandled exception, the framework will catch the exception and terminate the execution of the test case with a failure. However, the framework provides several macros to test whether a particular piece of code raises, or does not raise, exceptions. For more information, see the next recipe: *Asserting with Boost.Test*.

The test units that compose the module's test tree can be executed entirely or partially. In both cases, to execute the test units, execute the (binary) program that represents the test module. To execute only some of the test units, use the `--run_test` command-line option (or `--t` if you want to use a shorter name). This option allows you to filter the test units and specify either a path or label. A path consists of a sequence of test suite and/or test case names, such as `test_construction` or `test_operations/test_methods/test_offset`. A label is a name defined with the `label` decorator and is prefixed with `@` for the `run_test` parameter. This parameter is repeatable, which means you can specify multiple filters on it.

See also

- *Getting started with Boost.Test*
- *Asserting with Boost.Test*

Asserting with Boost.Test

A test case contains one or more tests. The `Boost.Test` library provides a series of APIs in the form of macros to write tests. In the previous recipe, you learned a bit about the `BOOST_TEST` macro, which is the most important and widely used macro of the library. In this recipe, we will discuss how it can be used in further detail.

Getting ready

You should now be familiar with writing test suites and test cases, a topic covered in the previous recipe.

How to do it…

The following list shows some of the most commonly used APIs for performing tests:

- `BOOST_TEST`, in its plain form, is used for most tests:

```
int a = 2, b = 4;
BOOST_TEST(a == b);

BOOST_TEST(4.201 == 4.200);

std::string s1{ "sample" };
std::string s2{ "text" };
BOOST_TEST(s1 == s2);
```

- `BOOST_TEST` along with the `tolerance()` manipulator are used to indicate the tolerance of floating point comparisons:

```
BOOST_TEST(4.201 == 4.200,
           boost::test_tools::tolerance(0.001));
```

- `BOOST_TEST` along with the `per_element()` manipulator are used to perform an element-wise comparison of containers (even of different types):

```
std::vector<int> v{ 1,2,3 };
std::list<short> l{ 1,2,3 };

BOOST_TEST(v == l, boost::test_tools::per_element());
```

- `BOOST_TEST` along with the ternary operator and compound statements using the logical `||` or `&&` require an extra set of parentheses:

```
BOOST_TEST((a > 0 ? true : false));
BOOST_TEST((a > 2 && b < 5));
```

- `BOOST_ERROR` is used to unconditionally fail a test and produce a message in the report. This is equivalent to `BOOST_TEST(false, message)`:

```
BOOST_ERROR("this test will fail");
```

- `BOOST_TEST_WARN` is used to produce a warning in the report in case a test is failing, without increasing the number of encountered errors and stopping the execution of the test case:

```
BOOST_TEST_WARN(a == 4, "something is not right");
```

- `BOOST_TEST_REQUIRE` is used to ensure that test case preconditions are met; the execution of the test case is stopped otherwise:

```
BOOST_TEST_REQUIRE(a == 4, "this is critical");
```

- `BOOST_FAIL` is used to unconditionally stop the execution of the test case, increase the number of encountered errors, and produce a message in the report. This is equivalent

to `BOOST_TEST_REQUIRE(false, message)`:

```
        BOOST_FAIL("must be implemented");
```

- `BOOST_IS_DEFINED` is used to check whether a particular preprocessor symbol is defined at runtime. It is used together with `BOOST_TEST` to perform validation and logging:

```
        BOOST_TEST(BOOST_IS_DEFINED(UNICODE));
```

How it works…

The library defines a variety of macros and manipulators for performing test assertions. The most commonly used one is `BOOST_TEST`. This macro simply evaluates an expression; if it fails, it increases the error count but continues the execution of the test case. It has three variants actually:

- `BOOST_TEST_CHECK` is the same as `BOOST_TEST` and is used to perform checks as described in the previous section.
- `BOOST_TEST_WARN` is used for assertions meant to provide information but without increasing the error count and stopping the execution of the test case.
- `BOOST_TEST_REQUIRE` is intended to ensure pre-conditions that are required for test cases to continue execution are met. Upon failure, this macro increases the error count and stops the execution of the test case.

The general form of the test macro is `BOOST_TEST(statement)`. This macro provides rich and flexible reporting capabilities. By default, it shows not only the statement, but also the value of the operands to enable quick identification of the failure's cause. However, the user could provide an alternative failure description; in this scenario, the message is logged in the test report:

```
BOOST_TEST(a == b);
// error: in "regular_tests": check a == b has failed [2 != 4]

BOOST_TEST(a == b, "not equal");
// error: in "regular_tests": not equal
```

This macro also allows you to control the comparison process with special support for the following:

- The first is floating point comparison, where tolerance can be defined to test equality.
- Secondly, it supports containers' comparison using several methods: default comparison (using the overloaded `operator==`), per-element comparison, and lexicographic comparison (using the lexicogrphical order). Per-element comparison enables the comparison of different types of containers (such as `vector` and `list`) in the order given by the forward iterators of the container; it also takes into account the size of the container (meaning that it first tests the sizes and, only if they are equal, it continues with the comparison of the elements).
- Lastly, it supports bitwise comparison of the operands. Upon failure, the framework reports the index of the bit where the comparison failed.

The `BOOST_TEST` macro does have some limitations. It cannot be used with compound statements that use a comma because such statements would be intercepted and handled by the preprocessor or the ternary operator, and compound statements using the logical operators `||` and `&&`. The latter cases have a workaround: a second pair of parentheses, as in `BOOST_TEST((statement))`.

Several macros are available for testing whether a particular exception is raised during the evaluation of an expression. In the following list, `<level>` is either `CHECK`, `WARN` , or `REQUIRE`:

- `BOOST_<level>_NO_THROW(expr)` checks whether an exception is raised from the `expr` expression. Any exception raised during the evaluation of `expr` is caught by this assertion and is not propagated to the test body. If any exception occurs, the assertion fails.
- `BOOST_<level>_THROW(expr, exception_type)` checks whether an exception of `exception_type` is raised from the `expr` expression. If the expression `expr` does not raise any exception, then the assertion fails. Exceptions of types other than `expression_type` are not caught by this assertion and could be propagated to the test body. Uncaught exceptions in a test case are caught by the execution monitor, but they result in failed test cases.
- `BOOST_<level>_EXCEPTION(expr, exception_type, predicate)` checks whether an expression of `expression_type` is raised from the `expr` expression; if so, it passes the expression to the predicate for further examination. If no exception is raised or an exception of a type different than `exception_type` is raised, then the assertion behaves like `BOOST_<level>_THROW`.

There’s more…

This recipe discusses only the most common APIs for testing and their typical usage. However, the library provides many more APIs. For further reference, check the online documentation. For version 1.63, refer to http://www.boost.org/doc/libs/1_63_0/libs/test/doc/html/index.html.

See also

- *Writing and invoking tests with Boost.Test*

Using fixtures in Boost.Test

The larger a test module is and the more similar test cases are, the more likely it is to have test cases that require the same setup, cleanup, and maybe the same data. A component that contains these is called a *test fixture* or *test context*. Boost.Test provides several ways to define test fixtures for a test case, test suite, or a module (globally). In this recipe, we will look at how fixtures work.

Getting ready

The examples in this recipe use the following classes and functions for specifying test unit fixtures:

```
struct standard_fixture
{
  standard_fixture()  {BOOST_TEST_MESSAGE("setup");}
  ~standard_fixture() {BOOST_TEST_MESSAGE("cleanup");}
  int n {42};
};

struct extended_fixture
{
  std::string name;
  int data;

  extended_fixture(std::string const & n = "") : name(n), data(0)
  {
    BOOST_TEST_MESSAGE("setup "+ name);
  }

  ~extended_fixture()
  {
    BOOST_TEST_MESSAGE("cleanup "+ name);
  }
};

void fixture_setup()
{
  BOOST_TEST_MESSAGE("fixture setup");
}

void fixture_cleanup()
{
  BOOST_TEST_MESSAGE("fixture cleanup");
}
```

How to do it…

Use the following methods to define test fixtures for one or multiple test units:

- To define a fixture for a particular test case, use the `BOOST_FIXTURE_TEST_CASE` macro:

```
BOOST_FIXTURE_TEST_CASE(test_case, extended_fixture)
{
  data++;
  BOOST_TEST(data == 1);
}
```

- To define a fixture for all the test cases in a test suite, use `BOOST_FIXTURE_TEST_SUITE`:

```
BOOST_FIXTURE_TEST_SUITE(suite1, extended_fixture)

BOOST_AUTO_TEST_CASE(case1)
{
  BOOST_TEST(data == 0);
}

BOOST_AUTO_TEST_CASE(case2)
{
  data++;
  BOOST_TEST(data == 1);
}

BOOST_AUTO_TEST_SUITE_END()
```

- To define a fixture for all the test units in a test suite, except for one or several test units, use `BOOST_FIXTURE_TEST_SUITE` and overwrite it to a particular test unit with `BOOST_FIXTURE_TEST_CASE` for a test case and `BOOST_FIXTURE_TEST_SUITE` for a nested test suite:

```
BOOST_FIXTURE_TEST_SUITE(suite2, extended_fixture)

BOOST_AUTO_TEST_CASE(case1)
{
  BOOST_TEST(data == 0);
}

BOOST_FIXTURE_TEST_CASE(case2, standard_fixture)
{
  BOOST_TEST(n == 42);
}

BOOST_AUTO_TEST_SUITE_END()
```

- To define more than a single fixture for a test case or test suite, use `boost::unit_test::fixture` with the `BOOST_AUTO_TEST_SUITE` and `BOOST_AUTO_TEST_CASE` macros:

```
BOOST_AUTO_TEST_CASE(test_case_multifix,
  * boost::unit_test::fixture<extended_fixture>
      (std::string("fix1"))
  * boost::unit_test::fixture<extended_fixture>
      (std::string("fix2"))
  * boost::unit_test::fixture<standard_fixture>())
{
  BOOST_TEST(true);
}
```

- To use free functions as setup and teardown operations in the case of a fixture, use `boost::unit_test::fixture`:

```
BOOST_AUTO_TEST_CASE(test_case_funcfix,
  * boost::unit_test::fixture(&fixture_setup,
                              &fixture_cleanup))
{
  BOOST_TEST(true);
}
```

- To define a fixture for the module, use `BOOST_GLOBAL_FIXTURE`:

```
BOOST_GLOBAL_FIXTURE(standard_fixture);
```

How it works…

The library supports several fixture models:

- A *class model*, where the constructor acts as the setup function and the destructor as the cleanup function. An extended model allows the constructor to have one parameter. In the preceding example, `standard_fixture` implemented the first model and `extended_fixture` the second model.
- A *pair of free functions*: one that defines the setup and the other, optional, that implements the cleanup code. In the preceding example, we came across these when discussing `fixture_setup()` and `fixture_cleanup()`.

Fixtures implemented as classes can also have data members, and these members are made available to the test unit. If a fixture is defined for a test suite, it is available implicitly to all the test units that are grouped under this test suite. However, it is possible that test units contained in such a test suite could redefine the fixture. In this case, the fixture defined in the closest scope is the one available to the test unit.

It is possible to define multiple fixtures for a test unit. However, this is done with the `boost::unit_test::fixture()` decorator, not with macros. The test suite and test case are defined in this case with the `BOOST_TEST_SUITE`/`BOOST_AUTO_TEST_SUITE` and `BOOST_TEST_CASE`/`BOOST_AUTO_TEST_CASE` macros. Multiple `fixture()` decorators can be composed together with `operator *`, as seen in the previous section. A drawback of this approach is that if you use the fixture decorator with a class that has member data, then these members will not be available for the test units.

A new fixture object is constructed for each test case when it is executed, and the object is destroyed at the end of the test case.

The fixture state is not shared among different test cases. Therefore, the constructor and destructor are called once for each test case. You must make sure these special functions do not contain code which is supposed to be executed only once per module. If this is the case, you should set a global fixture for the entire module.

A global fixture uses the generic test class model (the model with the default constructor); you can define any number of global fixtures (allowing you to organize setup and cleanup by category, if necessary). Global fixtures are defined with the `BOOST_GLOBAL_FIXTURE` macro, and they have to be defined at the test file scope (not inside any test unit).

See also

- *Writing and invoking tests with Boost.Test*

Controlling outputs with Boost.Test

The framework provides the ability to customize what is shown in the test log and test report and the format of the results. Currently, there are two supported: a human-readable format and XML (also with a JUNIT format for the test log). However, it is possible to create and add your own format. The configuration of what is shown in the output can be done both at runtime, through command-line switches, and at compile time, through various APIs. During the execution of the tests, the framework collects all the events in a log. At the end, it produces a report that represents a summary of the execution with different levels of details. In the case of a failure, the report contains detailed information about the location and the cause, including actual and expected values. This helps developers quickly identify the error. In this recipe, we will see how to control what is written in the log and the report and in which format; we do this using the command-line options at runtime.

Getting ready

For the examples presented in this recipe, we will use the following test module:

```
#define BOOST_TEST_MODULE Controlling output
#include <boost/test/included/unit_test.hpp>

BOOST_AUTO_TEST_CASE(test_case)
{
  BOOST_TEST(true);
}

BOOST_AUTO_TEST_SUITE(test_suite)

BOOST_AUTO_TEST_CASE(test_case)
{
  int a = 42;
  BOOST_TEST(a == 0);
}

BOOST_AUTO_TEST_SUITE_END()
```

How to do it…

To control the test log output, do the following:

- Use either the `--log_format=<format>` or `-f <format>` command-line option to specify the log format. The possible formats are `HRF` (the default value), `XML`, and `JUNIT`.
- Use either the `--log_level=<level>` or `-l <level>` command-line option to specify the log level. The possible log levels include `error` (default for HRF and XML), `warning`, `all`, or `success` (the default for JUNIT).
- Use either the `--log_sink=<stream or file name>` or `-k <stream or file name>` command-line option to specify the location where the framework should write the test log. The possible options are `stdout` (default for HRM and XML), `stderr`, or an arbitrary file name (default for JUNIT).

To control the test report output, do the following:

- Use either the `--report_format=<format>` or `-m <format>` command-line option to specify the report format. The possible formats are `HRF` (the default value) and `XML`.
- Use either the `--report_level=<format>` or `-r <format>` command-line option to specify the report level. The possible formats are `confirm` (the default value), `no` (for no report), `short`, and `detailed`.
- Use either the `--report_sink=<stream or file name>` or `-e <stream or file name>` command-line option to specify the location where the framework should write the report log. The possible options are `stderr` (the default value), `stdout`, or an arbitrary file name.

How it works…

When you run the test module from a console/terminal, you see both the test log and test report, with the test report following the test log. For the test module shown earlier, the default output is as follows. The first three lines represent the test log and the last line the test report:

```
Running 2 test cases...
f:/chapter11bt_05/main.cpp(14): error: in "test_suite/test_case":
check a == 0 has failed [42 != 0]

*** 1 failure is detected in the test module "Controlling output"
```

The content of both the test log and test report can be made available in several formats. The default is a human-readable format (or HRF); however, the framework also supports XML, and for the test log, the JUNIT format. This is a format intended for automated tools, such as continuous build or integration tools. Apart from these options, you can implement your own format for the test log by implementing your own class derived from `boost::unit_test::unit_test_log_formatter`. The next example shows how to format the test log (the first example) and the test report (the second example) using XML (each highlighted in bold):

```
chapter11bt_05.exe -f XML
<TestLog><Error file="f:/chapter11bt_05/main.cpp"
line="14"><![CDATA[check a == 0 has failed [42 != 0]]]>
</Error></TestLog>
*** 1 failure is detected in the test module "Controlling output"

chapter11bt_05.exe -m XML
Running 2 test cases...
f:/chapter11bt_05/main.cpp(14): error: in "test_suite/test_case":
check a == 0 has failed [42 != 0]
<TestResult><TestSuite name="Controlling output" result="failed"
assertions_passed="1" assertions_failed="1" warnings_failed="0"
expected_failures="0" test_cases_passed="1"
test_cases_passed_with_warnings="0" test_cases_failed="1"
test_cases_skipped="0" test_cases_aborted="0"></TestSuite>
</TestResult>
```

The log or report level represents the verbosity of the output. The possible values of the verbosity level of a log are shown in the following table, ordered from the lowest to the highest level. A higher level in the table includes all the messages of the levels above it.

Level	Messages that are reported
`nothing`	Nothing is logged
`fatal_error`	System or user fatal errors and all the messages describing failed assertions on the `REQUIRE` level (such as `BOOST_TEST_REQUIRE` and `BOOST_REQUIRE_`)
`system_error`	System non-fatal errors
`cpp_exception`	Uncaught C++ exceptions
`error`	Failed assertion on the `CHECK` level (`BOOST_TEST` and `BOOST_CHECK_`)
`warning`	Failed assertion on the `WARN` level (`BOOST_TEST_WARN` and `BOOST_WARN_`)
`message`	

	Messages generated by `BOOST_TEST_MESSAGE`
`test_suite`	Notification at the start and finish states of each test unit
`all` / `success`	All the messages, including passed assertions

The available formats of the test report are described in the following table:

Level	Description
`no`	No report is produced
`confirm`	**Passing test:** *** No errors detected **Skipped test:** *** The <name> test suite was skipped; see the standard output for details **Aborted test:** *** The <name> test suite was aborted; see the standard output for details **Failed test without failed assertions:** *** Errors were detected in the <name> test suite; see the standard output for details **Failed test:** *** N failures are detected in the <name> test suite **Failed test with some failures expected:** *** N failures are detected (M failures are expected) in the <name> test suite
`detailed`	Results are reported in a hierarchical fashion (each test unit is reported as part of the parent test unit), but only relevant information appears. Test cases that do not have failing assertions do not produce entries in the report. The test case/suite <name> has passed/was skipped/was aborted/has failed/ with: N assertions out of M passed N assertions out of M failed N warnings out of M failed X failures expected
`short`	Similar to detailed, but this reports information only to the master test suite

The standard output stream (`stdout`) is the default location where the test log is written, and the standard error stream (`stderr`) is the default location of the test report. However, both the test log and test report can be redirected to another stream or file. In addition to these options, it is possible to specify a separate file for reporting memory leaks, using the `--report_memory_leaks_to=<file name>` command-line option. If this option is not present and memory leaks are detected, they are reported to the standard error stream.

There’s more…

In addition to the options discussed in this recipe, the framework provides additional compile time APIs, for controlling the output. For a comprehensive description of these APIs as well as the features described in this recipe, check the framework documentation at http://www.boost.org/doc/libs/1_63_0/libs/test/doc/html/index.html.

See also

- *Writing and invoking tests with Boost.Test*

Getting started with Google Test

Google Test is one of the most used testing frameworks of C++. It enables developers to write unit tests on multiple platforms, using multiple compilers. Google Test is a portable, lightweight framework that has a simple, yet comprehensive API for writing tests using asserts; here, tests are grouped into test cases and test cases into test programs. The framework provides useful features, such as repeating a test a number of times and breaking a test to invoke the debugger at the first failure. Its assertions work regardless of whether exceptions are enabled or not. The next recipe will cover the most important features of the framework. This recipe will show you how to install the framework and set up your first testing project.

Getting ready

The Google Test framework, just like Boost.Test, has a macro-based API. Although you only need to use the supplied macros for writing tests, a good understanding of macros is recommended in order to use the framework well.

How to do it…

In order to set up your environment to use Google Test, do the following:

1. Clone or download the Git repository from https://github.com/google/googletest.
2. Once you download the repository, unzip the content of the archive.
3. Build the framework using the provided build scripts.

To create your first test program using Google Test, do the following:

1. Create a new empty C++ project.
2. Do the necessary setup specific to the development environment you are using to make the framework's *headers* folder available to the project for including header files.
3. Link the project to the *gtest* shared library.
4. Add a new source file to the project with the following content:

```
#include <gtest/gtest.h>
TEST(FirstTestCase, FirstTestFunction)
{
  ASSERT_TRUE(true);
}

int main(int argc, char **argv)
{
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

5. Build and run the project.

How it works…

The Google Test framework provides a simple and easy-to-use set of macros for creating tests and writing assertions. The test's structure is also simplified compared to other testing frameworks, such as Boost.Test. Test functions are grouped into test cases and test cases into test programs. It is important to notice that a test function in Google Test is equivalent to a test case in Boost.Test and other frameworks, and a test case in Google Test is equivalent to a test suite in Boost.Test. However, test cases in Google Test cannot contain other test cases, but only test functions. The framework provides a rich set of assertions, both fatal and non-fatal, great support for exception handling, and the ability to customize the way tests are executed and how the output should be generated. Documentation on this framework is available on GitHub's project page. The sample code shown in the previous section contains the following parts:

1. `#include <gtest/gtest.h>` includes the main header of the framework.
2. `TEST(FirstTestCase, FirstTestFunction)` declares a test function called `FirstTestFunction` as part of a test case called `FirstTestCase`. A test function has no arguments and returns `void`. Multiple test functions can be grouped with the same test case.
3. `ASSERT_TRUE(true);` is an assertion macro that yields a fatal error and returns from the current function in case the condition evaluates to `false`. The framework defines many more assertion macros, which we will see in the *Asserting with Google Test* recipe.
4. `::testing::InitGoogleTest(&argc, argv);` initializes the framework and must be called before `RUN_ALL_TESTS()`.
5. `return RUN_ALL_TESTS();` automatically detects and calls all the tests defined with either the `TEST()` or `TEST_F()` macro. The return value returned from the macro is used as the return value of the `main()` function. This is important, because the automated testing service determines the result of a test program according to the value returned from the `main()` function, not the output printed to the `stdout` or `stderr` streams. The `RUN_ALL_TESTS()` macro must be called only once; calling it multiple times is not supported because it conflicts with some advanced features of the framework.

Executing this test program will provide the following result:

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from FirstTestCase
[ RUN ] FirstTestCase.FirstTestFunction
[ OK ] FirstTestCase.FirstTestFunction (0 ms)
[----------] 1 test from FirstTestCase (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (2 ms total)
[ PASSED ] 1 test.
```

For many test programs, the content of the `main()` function is identical to the one shown in this recipe, in the example from the *How to do it…* section. To avoid writing such a `main()` function, the framework provides a basic implementation that you can use by linking your program with the `gtest_main` shared library.

There’s more…

The Google Test framework can also be used with other testing frameworks. You can write tests using another testing framework, such as Boost.Test or CppUnit, and use the Google Test assertion macros. To do so, set the `throw_on_failure` flag, either from the code or command line, with the `--gtest_throw_on_failure` argument. Alternatively, use the `GTEST_THROW_ON_FAILURE` environment variable and initialize the framework, as shown in the following code snippet:

```
#include "gtest/gtest.h"

int main(int argc, char** argv)
{
  ::testing::GTEST_FLAG(throw_on_failure) = true;
  ::testing::InitGoogleTest(&argc, argv);
}
```

When you enable the `throw_on_failure` option, assertions that fail will print an error message and throw an exception, which would be caught by the host testing framework and treated as a failure. If exceptions are not enabled, then a failed Google Test assertion will indicate your program to exit with a non-zero code, which again will be treated as a failure by the host testing framework.

See also

- *Writing and invoking tests with Google Test*
- *Asserting with Google Test*

Writing and invoking tests with Google Test

In the previous recipe, we had a glimpse of what it takes to write simple tests with the Google Test framework. Multiple tests can be grouped into a test case and one or more test cases grouped into a test program. In this recipe, we will see how to create and run tests.

Getting ready

For the sample code in this recipe, use the `point3d` class discussed in the *Writing and invoking tests with Boost.Test* recipe.

How to do it…

Use the following macros to create tests:

- `TEST(TestCaseName, TestName)` defines a test called `TestName` as part of a test case called `TestCaseName`:

```
TEST(TestConstruction, TesConstructor)
{
  auto p = point3d{ 1,2,3 };
  ASSERT_EQ(p.x(), 1);
  ASSERT_EQ(p.x(), 2);
  ASSERT_EQ(p.x(), 3);
}

TEST(TestConstruction, TestOrigin)
{
  auto p = point3d::origin();
  ASSERT_EQ(p.x(), 0);
  ASSERT_EQ(p.x(), 0);
  ASSERT_EQ(p.x(), 0);
}
```

- `TEST_F(TestCaseWithFixture, TestName)` defines a test called `TestName` as part of a test case, using a fixture called `TestCaseWithFixture`. You'll find details about how this works in the *Using test fixtures with Google Test* recipe.

To execute the tests, do the following:

- Use the `RUN_ALL_TESTS()` macro to run all the tests defined in the test program. This must be called only once from the `main()` function after the framework has been initialized.
- Use the `--gtest_filter=<filter>` command-line option to filter the tests to run.
- Use the `--gtest_repeat=<count>` command-line option to repeat the selected tests the specified number of times.
- Use the `--gtest_break_on_failure` command-line option to attach the debugger to debug the test program when the first test fails.

How it works…

There are several macros available for defining tests (as part of a test case). The most common ones are `TEST` and `TEST_F`. The latter is used with fixtures, which will be discussed in detail in a later recipe. Other macros for defining tests are `TYPED_TEST` for writing typed tests and `TYPED_TEST_P` for writing type-parameterized tests. However, these are more advanced topics and are beyond the scope of this book. The `TEST` and `TEST_F` macros take two arguments: the first is the name of the test case and the second is the name of the test. These two form the full name of a test, and they must be valid C++ identifiers; they should not contain underscores, though. Different test cases can contain tests with the same name (because the full name is still unique). Both the macros automatically register the tests with the framework; therefore, no explicit input is required from the user to do this.

A test can either fail or succeed. A test fails if an assertion fails or an uncaught exception occurs. Except for these two instances, the test always succeeds.

To invoke the test, call `RUN_ALL_TESTS()`, but you can do this only once in a test program and only after the framework has been initialized with a call to `::testing::InitGoogleTest()`. This macro runs all the tests in the test program. However, it is possible that you select only some tests to run. You can do this either by setting up an environment variable called `GTEST_FILTER` with the appropriate filter, or passing the filter as a command-line argument with the `--gtest_filter` flag. If any of these two are present, the framework only runs the tests whose full name matches the filter. The filter may include wildcards: `*` to match any string and `?` to match any character. Negative patterns (what should be omitted) are introduced with a hyphen (`-`). The following are examples of filters:

Filter	Description
`--gtest_filter=*`	Run all the tests
`--gtest_filter=TestConstruction.*`	Run all the tests from the test case called *TestConstruction*
`--gtest_filter=TestOperations.*-TestOperations.TestLess`	Run all the tests from the test case called *TestOperations*, except for a test called *TestLess*
`--gtest_filter=*Operations*:*Construction*`	Run all the tests whose full name contains either *Operations* or *Construction*

The following listing is the output of a test program containing the tests shown earlier when invoked with the command-line argument `--gtest_filter=TestConstruction.*-TestConstruction.TestConstructor`:

```
Note: Google Test filter = TestConstruction.*-TestConstruction.TestConstructor
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from TestConstruction
[ RUN ] TestConstruction.TestOrigin
[ OK ] TestConstruction.TestOrigin (0 ms)
[----------] 1 test from TestConstruction (3 ms total)

[----------] Global test environment tear-down
```

```
[==========] 1 test from 1 test case ran. (6 ms total)
[ PASSED ] 1 test.
```

See also

- *Getting started with Google Test*
- *Asserting with Google Test*
- *Using test fixtures with Google Test*

Asserting with Google Test

The Google Test framework provides a rich set of both fatal and non-fatal assertion macros, which resemble function calls, to verify the tested code. When these assertions fail, the framework displays the source file, line number, and relevant error message (including custom error messages) to help developers quickly identify the failed code. We have already seen some simple examples on how to use the `ASSERT_TRUE` macro; in this recipe, we will look at other available macros.

How to do it…

Use the following macros to verify the tested code:

- Use `ASSERT_TRUE(condition)` or `EXPECT_TRUE(condition)` to check whether the condition is `true` and `ASSERT_FASE(condition)` or `EXPECT_FALSE(condition)` to check whether the condition is `false`, as shown in the following code:

```
EXPECT_TRUE(2 + 2 == 2 * 2);
EXPECT_FALSE(1 == 2);

ASSERT_TRUE(2 + 2 == 2 * 2);
ASSERT_FALSE(1 == 2);
```

- Use `ASSERT_XX(val1, val2)` or `EXPECT_XX(val1, val2)` to compare the two values, where `XX` is one of the following: `EQ(val1 == val2)`, `NE(val1 != val2)`, `LT(val1 < val2)`, `LE(val1 <= val2)`, `GT(val1 > val2)`, or `GE(val1 >= val2)`. This is illustrated in the following code:

```
auto a = 42, b = 10;
EXPECT_EQ(a, 42);
EXPECT_NE(a, b);
EXPECT_LT(b, a);
EXPECT_LE(b, 11);
EXPECT_GT(a, b);
EXPECT_GE(b, 10);
```

- Use `ASSERT_STRXX(str1, str2)` or `EXPECT_STRXX(str1, str2)` to compare the two null-terminated strings, where `XX` is one of the following: `EQ` (the strings have the same content), `NE` (the strings don't have the same content), `CASEEQ` (the strings have the same content with the case ignored), and `CASENE` (the strings don't have the same content with the case ignored). This is illustrated in the following code snippet:

```
auto str = "sample";
EXPECT_STREQ(str, "sample");
EXPECT_STRNE(str, "simple");
ASSERT_STRCASEEQ(str, "SAMPLE");
ASSERT_STRCASENE(str, "SIMPLE");
```

- Use `ASSERT_FLOAT_EQ(val1, val2)` or `EXPECT_FLOAT_EQ(val1, val2)` to check whether the two `float` values are almost equal and `ASSERT_DOUBLE_EQ(val1, val2)` or `EXPECT_DOUBLE_EQ(val1, val2)` to check whether the two `double` values are almost equal; they should differ by at most 4 ULP (*units in the last place*). Use `ASSERT_NEAR(val1, val2, abserr)` or `ASSERT_NEAR(val1, val2, abserr)` to check whether the difference between the two values is not greater than the specified absolute value:

```
EXPECT_FLOAT_EQ(1.9999999f, 1.9999998f);
ASSERT_FLOAT_EQ(1.9999999f, 1.9999998f);
```

- Use `ASSERT_THROW(statement, exception_type)` or `EXPECT_THROW(statement, exception_type)` to check whether the statement throws an exception of the specified type, `ASSERT_ANY_THROW(statement)` or `EXPECT_ANY_THROW(statement)` to check whether the statement throws an exception of any type, and `ASSERT_NO_THROW(statement)` or

`EXPECT_NO_THROW(statement)` to check whether the statement throws any exception or not:

```
void function_that_throws()
{
  throw std::runtime_error("error");
}

void function_no_throw()
{
}

EXPECT_THROW(function_that_throws(),
             std::runtime_error);
EXPECT_ANY_THROW(function_that_throws());
EXPECT_NO_THROW(function_no_throw());

ASSERT_THROW(function_that_throws(),
             std::runtime_error);
ASSERT_ANY_THROW(function_that_throws());
ASSERT_NO_THROW(function_no_throw());
```

- Use `ASSERT_PRED1(pred, val)` or `EXPECT_PRED1(pred, val)` to check whether `pred(val)` returns `true`, `ASSERT_PRED2(pred, val1, val2)` or `EXPECT_PRED2(pred, val1, val2)` to check whether `pred(val1, val2)` returns `true`, and so on; use this for n-ary predicate functions or functors:

```
bool is_positive(int const val)
{
  return val != 0;
}

bool is_double(int const val1, int const val2)
{
  return val2 + val2 == val1;
}

EXPECT_PRED1(is_positive, 42);
EXPECT_PRED2(is_double, 42, 21);

ASSERT_PRED1(is_positive, 42);
ASSERT_PRED2(is_double, 42, 21);
```

- Use `ASSERT_HRESULT_SUCCEEDED(expr)` or `EXPECT_HRESULT_SUCCEEDED(expr)` to check whether `expr` is a success `HRESULT` and `ASSERT_HRESULT_FAILED(expr)` or `EXPECT_HRESULT_FAILED(expr)` to check whether `expr` is a failure `HRESULT`. These assertions are intended to be used on Windows.
- Use `FAIL()` to generate a fatal failure and `ADD_FAILURE()` or `ADD_FAILURE_AT(filename, line)` to generate non-fatal failures:

```
ADD_FAILURE();
ADD_FAILURE_AT(__FILE__, __LINE__);
```

How it works…

All the asserts are available in two versions:

- `ASSERT_*`: This generates fatal failures, preventing further execution of the current test function.
- `EXPECT_*`: This generates non-fatal failures, which means that the execution of the test function continues even if the assertion fails.

Use the `EXPECT_*` assertion if not meeting the condition is not a critical error or if you want the test function to continue in order to get as many error messages as possible. In other cases, use the `ASSERT_*` version of the test assertions.

You will find details about the assertions presented here in the framework's online documentation, which is available on GitHub; this is where the project is located. A special note on floating point comparison is, however, necessary. Due to round-offs (fractional part cannot be represented as a finite sum of the inverse powers of two), floating point values do not match exactly. Therefore comparison should be done within a relative error bound. The macros `ASSERT_EQ`/`EXPECT_EQ` are not suitable for comparing floating points, and the framework provides another set of assertions. `ASSERT_FLOAT_EQ`/`ASSERT_DOUBLE_EQ` and `EXPECT_FLOAT_EQ`/`EXPECT_DOUBLE_EQ` perform a comparison with a default error of 4ULP.

ULP is a unit of measurement for the spacing between floating point numbers, that is, the value the least significant digit represents if it is 1. For more information on this, read the Comparing Floating Point Numbers, 2012 Edition article by Bruce Dawson: https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/.

See also

- *Writing and invoking tests with Google Test*

Using text fixtures with Google Test

The framework provides support for using fixtures as reusable components for all the tests that are part of a test case. It also provides support for setting up the global environment in which the tests would run. In this recipe, you will find stepwise instructions on how to define and use test fixtures and also set up the test environment.

Getting ready

You should now be familiar with writing and invoking tests using the Google Test framework, a topic which was covered earlier in this chapter, specifically in the *Writing and invoking tests with Google Test* recipe.

How to do it…

To create and use a test fixture, do the following:

1. Create a class derived from the `::testing::Test` class:

```
class TestFixture : public ::testing::Test
{
};
```

2. Use the constructor to initialize the fixture and the destructor to clean it up:

```
protected:
  TestFixture()
  {
    std::cout << "constructing fixture" << std::endl;
    data.resize(10);
    std::iota(std::begin(data), std::end(data), 1);
  }

  ~TestFixture()
  {
    std::cout << "destroying fixture" << std::endl;
  }
```

3. Alternatively, you can override the virtual methods `SetUp()` and `TearDown()` for the same purpose.
4. Add member data and functions to the class to make them available to the tests:

```
protected:
  std::vector<int> data;
```

5. Use the `TEST_F` macro to define tests using fixtures, and specify the fixture class name as the test case name:

```
TEST_F(TestFixture, TestData)
{
  ASSERT_EQ(data.size(), 10);
  ASSERT_EQ(data[0], 1);
  ASSERT_EQ(data[data.size()-1], data.size());
}
```

To customize the setting up of the environment for running tests, do the following:

1. Create a class derived from `::testing::Environment`:

```
class TestEnvironment : public ::testing::Environment
{
};
```

2. Override the virtual methods `SetUp()` and `TearDown()` to perform setup and cleanup operations:

```
public:
  virtual void SetUp() override
  {
    std::cout << "environment setup" << std::endl;
  }
```

```
  virtual void TearDown() override
  {
    std::cout << "environment cleanup" << std::endl;
  }

  int n{ 42 };
```

3. Register the environment with a call to `::testing::AddGlobalTestEnvironment()` before calling `RUN_ALL_TESTS()`:

```
int main(int argc, char **argv)
{
  ::testing::InitGoogleTest(&argc, argv);
  ::testing::AddGlobalTestEnvironment(new TestEnvironment{});
  return RUN_ALL_TESTS();
}
```

How it works…

Text fixtures enable users to share data configurations between multiple tests. Fixture objects are not shared between tests. A different fixture object is created for each test that is associated with the text function. The following operations are performed by the framework for each test coming from a fixture:

1. Create a new fixture object.
2. Call its `SetUp()` virtual method.
3. Run the test.
4. Call the fixture's `TearDown()` virtual method.
5. Destroy the fixture object.

You can set up and clean the fixture objects in two ways: using the constructor and destructor or the pair of `SetUp()` and `TearDown()` virtual methods. For most cases, the former way is preferred. The use of virtual methods is suitable in several cases, though:

- When the tear-down operation throws an exception, as exceptions are not allowed to leave destructors.
- If you are required to use assertion macros during cleanup and you use the `--gtest_throw_on_failure` flag which determines the macros to be thrown upon a failure.
- If you need to call virtual methods (which might be overridden in a derived class), as virtual calls should not be invoked from the constructor or destructor.

Tests which use fixtures must be defined using the `TEST_F` macro (where `_F` stands for fixture). Trying to declare them using the `TEST` macro generates compiler errors.

The environments in which tests are run can also be customized. The mechanism is similar to test fixtures: you derive the base `testing::Environment` class and override the `SetUp()` and `TearDown()` virtual functions. Instances of these derived environment classes must be registered with the framework with a call to `testing::AddGlobalTestEnvironment()`; however, this has to be done before you run the tests. You can register as many instances as you want, in which case the `SetUp()` method is called for the objects in the order they were registered and the `TearDown()` method in reverse order. You must pass dynamically instantiated objects to this function. The framework takes ownership of the objects and deletes them before the program terminates; therefore, do not delete them yourselves.

Environment objects are neither available to the tests, nor intended for providing data to the tests. Their purpose is to customize the global environment for running the tests.

See also

- *Writing and invoking tests with Google Test*

Controlling output with Google Test

By default, the output of a Google Test test program goes to the standard stream, printed in a human-readable form. The framework provides several options for customizing the output, including printing XML to a disk file in a JUNIT-based format. This recipe will explore the options available to control the output.

Getting ready

For the purpose of this recipe, let’s consider the following test program:

```
    #include <gtest/gtest.h>

    TEST(Sample, Test)
    {
      auto a = 42;
      ASSERT_EQ(a, 0);
    }

    int main(int argc, char **argv)
    {
      ::testing::InitGoogleTest(&argc, argv);
      return RUN_ALL_TESTS();
    }
```

Its output is as follows:

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from Sample
[ RUN ] Sample.Test
f:/chapter11gt_05/main.cpp(6): error: Expected: a
  Which is: 42
To be equal to: 0
[ FAILED ] Sample.Test (1 ms)
[----------] 1 test from Sample (1 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (2 ms total)
[ PASSED ] 0 tests.
[ FAILED ] 1 test, listed below:
[ FAILED ] Sample.Test

1 FAILED TEST
```

How to do it…

To control the output of a test program, you can:

- Use the `--gtest_output` command-line option or the `GTEST_OUTPUT` environment variable with the `xml:filepath` string to specify the location of a file where the XML report is to be written:

```
chapter11gt_05.exe --gtest_output=xml:report.xml

<?xml version="1.0" encoding="UTF-8"?>
<testsuites tests="1" failures="1"
            disabled="0" errors="0"
            timestamp="2017-02-25T00:02:27"
            time="0.006" name="AllTests">
 <testsuite name="Sample" tests="1"
            failures="1" disabled="0"
            errors="0" time="0.003">
    <testcase name="Test" status="run" time="0.002"
              classname="Sample">
      <failure message="f:/chapter11gt_05/main.cpp:6
 Expected: a
 Which is: 42
 To be equal to: 0" type="">
        <![CDATA[f:/chapter11gt_05/main.cpp:6
              Expected: a
              Which is: 42
         To be equal to: 0]]></failure>
    </testcase>
  </testsuite>
</testsuites>
```

- Use the `--gtest_color` command-line option or the `GTEST_COLOR` environment variable and specify either `auto`, `yes`, or `no` to indicate whether the report should be printed to a terminal using colors or not:

```
chapter11gt_05.exe --gtest_color=no
```

- Use the `--gtest_print_time` command-line option or the `GTEST_PRINT_TIME` environment variable with the value `0` to suppress the printing time each test takes to execute:

```
chapter11gt_05.exe --gtest_print_time=0

[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from Sample
[ RUN ] Sample.Test
f:/chapter11gt_05/main.cpp(6): error: Expected: a
      Which is: 42
To be equal to: 0
[ FAILED ] Sample.Test
[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran.
[ PASSED ] 0 tests.
[ FAILED ] 1 test, listed below:
[ FAILED ] Sample.Test
```

How it works…

Generating a report in an XML format does not affect the human-readable report printed to the terminal. The output path can indicate either a file, a directory (in which case a file with the name of the executable is created—if it already exists from a previous run, it creates a file with a new name by suffixing it with a number), or nothing, in which case the report is written to a file called `test_detail.xml` in the current directory.

The XML report format is based on the JUnitReport Ant task and contains the following main elements:

- `<testsuites>`: This is the root element and it corresponds to the entire test program.
- `<testsuite>`: This corresponds to a test case, as Google Test test cases are equivalent to test suites in other frameworks.
- `<testcase>`: This corresponds to a test function, as Google Test test functions are equivalent to test cases in other frameworks.

By default, the framework reports the time it takes for each test to execute. This feature can be suppressed using the `--gtest_print_time` command-line option or the `GTEST_PRINT_TIME` environment variable, as shown earlier. This option was the default up to version 1.3.0.

See also

- *Writing and invoking tests with Google Test*
- *Using test fixtures with Google Test*

Getting started with Catch

Catch is a multiparadigm header-only testing framework for C++ and Objective-C. The name Catch stands for *C++ Automated Test Cases in Headers*. It enables developers to write tests using either the traditional style of test functions grouped in test cases or the *Behavior Driven Development (BDD)* style with *given-when-then* sections. Tests are self-registered and the framework provides several assertion macros; out of these, two are most used: one fatal, namely `REQUIRE`, and one non-fatal, namely `CHECK`. They perform expression decomposition of both left- and right-hand side values which are logged in case of failure.

Getting ready

The Catch test framework has a macro-based API. Although you only need to use the supplied macros for writing tests, a good understanding of macros is recommended if you want to use the framework well.

How to do it…

In order to set up your environment to use the Catch testing framework, do the following:

1. Clone or download the Git repository from https://github.com/philsquared/Catch.
2. Once you download the repository, unzip the content of the archive.

To create your first test program using Catch, do the following:

1. Create a new empty C++ project.
2. Do the necessary setup specific to the development environment you are using to make the framework's headers folder available to the project for including header files.
3. Add a new source file to the project with the following content:

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

TEST_CASE("first_test_case", "[learn][catch]")
{
  SECTION("first_test_function")
  {
    auto i{ 42 };
    REQUIRE(i == 42);
  }
}
```

4. Build and run the project.

How it works…

Catch enables developers to write test cases as self-registered functions; they can even provide a default implementation for the `main()` function so that you can focus on testing code and writing less setup code. Test cases are divided into sections that are run in isolation. The framework does not adhere to the style of the *setup-test-teardown* architecture. Instead, the test case sections (or rather the innermost ones, since sections can be nested) are the units of testing executed along with their enclosing sections. This makes the need for fixtures obsolete, because data and setup and tear-down code can be reused on multiple levels.

Test cases and sections are identified using strings, not identifiers (as in most testing frameworks). Test cases can also be tagged so that tests can be executed or listed based on tags. Test results are printed in a textual human-readable form; however, they can also be exported to XML, using either a Catch-specific schema or a JUNIT ANT schema for easy integration with continuous delivery systems. The execution of the tests can be parameterized to break upon failure (on Windows and Mac) so that you can attach a debugger and inspect the program.

The framework is easy to install and does not require compilation. The entire code is provided in the header files. There are two alternatives: a single header file or a collection of header files that include each other. In both cases, the only header file you have to include in your test program is `catch.hpp`.

The sample code shown in the previous section has the following parts:

1. `#define CATCH_CONFIG_MAIN` defines a macro that instructs the framework to provide a default implementation of the `main()` function.
2. `#include "catch.hpp"` includes the main header of the library (which, in turn, includes other headers).
3. `TEST_CASE("first_test_case", "[learn][catch]")` defines a test case called `first_test_case`, which has several associated tags: `learn` and `catch`. Tags are used to select to either run or just list test cases. Multiple test cases can be tagged with the same tags.
4. `SECTION("first_test_function")` defines a section, that is, a test function, called `first_test_function`, as part of the outer test case.
5. `REQUIRE(i == 42);` is an assertion that indicates the test to fail if the condition is not satisfied.

The output of running this program is as follows:

```
===============================================================
All tests passed (1 assertions in 1 test cases)
```

There’s more…

As mentioned before, the framework enables us to write tests using the BDD style with *give-when-then* sections. This was made possible using several aliases: `SCENARIO` for `TEST_CASE` and `GIVE`, `WHEN`, `AND_WHEN`, `THEN`, and `AND_THEN` for `SECTION`. Using this style, we can rewrite the test shown earlier as follows:

```
    SCENARIO("first_scenario", "[learn][catch]")
    {
      GIVEN("an integer")
      {
        auto i = 0;
        WHEN("assigned a value")
        {
          i = 42;
          THEN("the value can be read back")
          {
            REQUIRE(i == 42);
          }
        }
      }
    }
```

When executed successfully, the program prints the following output:

```
===============================================================
All tests passed (1 assertions in 1 test cases)
```

However, upon failure (let’s suppose we got the wrong condition: `i == 0`), the description provided for the scenario and the sections that failed is printed along with the expression that failed and the values on the left- and right-hand sides, as shown in the following snippet:

```
-------------------------------------------------------------
Scenario: first_scenario
 Given: an integer
 When: assigned a value
 Then: the value can be read back
-------------------------------------------------------------
 f:\chapter11ca_01\main.cpp(21)
.............................................................

 f:\chapter11ca_01\main.cpp(23): FAILED:
 REQUIRE( i == 0 )
 with expansion:
  42 == 0
```

See also

- *Writing and invoking tests with Catch*
- *Asserting with Catch*

Writing and invoking tests with Catch

The Catch framework enables you to write tests using either the traditional style of test cases and test functions or the BDD style with scenarios and *given-when-then* sections. Tests are defined as separate sections of a test case and can be nested as deep as you want. Whichever style you prefer, tests are defined with only two base macros. This recipe will show what these macros are and how they work.

How to do it…

To write tests using the traditional style, with test cases and test functions, do this:

- Use the `TEST_CASE` macro to define a test case with a name (as a string), and optionally, a list of its associated tags:

```
TEST_CASE("test construction", "[create]")
{
  // define sections here
}
```

- Use the `SECTION` macro to define a test function inside a test case, with `name` as a string:

```
TEST_CASE("test construction", "[create]")
{
  SECTION("test constructor")
  {
    auto p = point3d{ 1,2,3 };
    REQUIRE(p.x() == 1);
    REQUIRE(p.y() == 2);
    REQUIRE(p.z() == 4);
  }
}
```

- Define nested sections if you want to reuse the setup and teardown code or organize your tests in a hierarchical structure:

```
TEST_CASE("test operations", "[modify]")
{
  SECTION("test methods")
  {
    SECTION("test offset")
    {
      auto p = point3d{ 1,2,3 };
      p.offset(1, 1, 1);
      REQUIRE(p.x() == 2);
      REQUIRE(p.y() == 3);
      REQUIRE(p.z() == 3);
    }
  }
}
```

To write tests using the BDD style, do this:

- Define scenarios using the `SCENARIO` macro, specifying a name for it:

```
SCENARIO("modify existing object")
{
  // define sections here
}
```

- Define nested sections inside the scenario using the `GIVEN`, `WHEN`, and `THEN` macros, specifying a name for each one of them:

```
SCENARIO("modify existing object")
{
  GIVEN("a default constructed point")
  {
    auto p = point3d{};
    REQUIRE(p.x() == 0);
```

```
        REQUIRE(p.y() == 0);
        REQUIRE(p.z() == 0);

        WHEN("increased with 1 unit on all dims")
        {
          p.offset(1, 1, 1);

          THEN("all coordinates are equal to 1")
          {
            REQUIRE(p.x() == 1);
            REQUIRE(p.y() == 1);
            REQUIRE(p.z() == 1);
          }
        }
      }
    }
```

To execute the tests, do the following:

- To execute all the tests from your program (except hidden ones), run the test program without any command-line arguments (from the ones described in the following code).
- To execute only a specific set of test cases, provide a filter as a command-line argument. This can contain test case names, wildcards, tag names, and tag expressions:

```
chapter11ca_02.exe "test construction"

test construction
   test constructor
-----------------------------------------------
f:\chapter11ca_02\main.cpp(7)
...............................................
f:\chapter11ca_02\main.cpp(12): FAILED:
  REQUIRE( p.z() == 4 )
with expansion:
  3 == 4

===============================================
test cases: 1 | 1 failed
assertions: 6 | 5 passed | 1 failed
```

- To execute only a particular section (or set of sections), use the command-line argument `--section` or `-c` with the section name (can be used multiple times for multiple sections):

```
chapter11ca_02.exe "test construction" --section "test origin"
===============================================
All tests passed (3 assertions in 1 test case)
```

- To specify the order in which test cases should be run, use the command-line argument `--order` with one of the values: `decl` (for the order of declaration), `lex` (for a lexicographic ordering by the name), or `rand` (for a random order determined with `std::random_shuffle()`). Here's an illustration of this:

```
chapter11ca_02.exe --order lex
```

How it works…

Test cases are self-registered and do not require any additional work from the developer to set the test program, other than defining the test cases and test functions. Tests functions are defined as sections of test cases (using the `SECTION` macro), and they can be nested. There is no limit to the depth of section nesting. Test cases and test functions, which further will be referred to as sections, form a tree structure, with the test cases on the root nodes and the most inner sections as leafs. When the test program runs, it is the leaf sections that are executed. Each leaf section is executed in isolation of the other leaf section. However, the execution path starts at the root test case and continues downward toward the innermost section. All of the code encountered on the path is executed entirely for each run. This means that when multiple sections share common code (from a parent section or the test case), the same code is executed once for each section, without any data being shared between executions. This has the effect that it eliminates the need for a special fixture approach on one hand. On the other hand, it enables multiple fixtures for each section (everything that is encountered up in the path), a feature that many testing frameworks lack.

The BDD style of writing test cases is powered by the same two macros, namely `TEST_CASE` and `SECTION`, and the ability to test sections. In fact, the macro `SCENARIO` is a redefinition of `TEST_CASE` and `GIVEN`, `WHEN`, `AND_WHEN`, `THEN`, and `AND_THEN` are redefinitions of `SECTION`:

```
    #define SCENARIO(...) TEST_CASE("Scenario: " __VA_ARGS__)
    #define GIVEN(desc) SECTION(std::string(" Given: ") +
                                desc, "")
    #define WHEN(desc) SECTION(std::string(" When: ") +
                                desc, "")
    #define AND_WHEN(desc) SECTION(std::string("And when: ") +
                                desc, "")
    #define THEN(desc) SECTION(std::string(" Then: ") +
                                desc, "")
    #define AND_THEN(desc) SECTION(std::string(" And: ") +
                                desc, "")
```

When you execute a test program, all defined tests are run. This, however, excludes hidden tests, which are specified either using a name that starts with `./` or a tag that starts with a period. It is possible to force the running of hidden tests too by providing the command-line argument `[.]` or `[hide]`.

It is possible to filter the test cases to execute. This can be done using either the name or the tags. The following table displays some of the possible options:

Argument	Description
`"test construction"`	The test case called *test construction*
`test*`	All test cases that start with *test*
`~"test construction"`	All test cases, except the one called *test construction*
`~*equal*`	All test cases, except those that contain the word *equal*
`[modify]`	All test cases tagged with `[modify]`

`[modify],[compare][op]`	All test cases that are tagged with either `[modify]` or both `[compare]` and `[op]`

The execution of particular test functions is also possible by specifying one or more section names with the command-line argument `--section` or `-c`. If you do so, be aware that the entire test path from the root test case to the selected section will be executed. Moreover, if you do not specify a test case or a set of test cases first, then all the test cases will be executed, though only the matching sections within them.

See also

- *Getting started with Catch*
- *Asserting with Catch*

Asserting with Catch

Unlike other testing frameworks, Catch does not provide a large set of assertion macros. It has two main macros: `REQUIRE`, which produces a fatal error stopping the execution of the test case upon failure and `CHECK`, which produces a non-fatal error upon failure, continuing the execution of the test case. Several additional macros are defined; in this recipe, we will see how to put them to work.

Getting ready

You should now be familiar with writing test cases and test functions using Catch, a topic covered in the previous recipe.

How to do it…

The following list contains the available options for asserting with the Catch framework:

- Use `CHECK(expr)` to check whether `expr` evaluates to `true`, continuing the execution in case of failure, and `REQUIRE(expr)` to make sure that `expr` evaluates to `true` and stop the execution of the test in case of failure:

```
int a = 42;
CHECK(a == 42);
REQUIRE(a == 42);
```

- Use `CHECK_FALSE(expr)` and `REQUIRE_FALSE(expr)` to make sure that `expr` evaluates to `false` and produce either a non-fatal or fatal error in case of failure:

```
int a = 42;
CHECK_FALSE(a > 100);
REQUIRE_FALSE(a > 100);
```

- Use the `Approx` class to compare floating point values with a given approximation. The method `epsilon()` sets a maximum percentage (as a value between 0 and 1) by which the value can differ:

```
double a = 42.5;
CHECK(42.0 == Approx(a).epsilon(0.02));
REQUIRE(42.0 == Approx(a).epsilon(0.02));
```

- Use `CHECK_NOTHROW(expr)`/`REQUIRE_NOTHROW(expr)` to verify that `expr` does not throw any error, `CHECK_THROWS(expr)`/`REQUIRE_THROWS(expr)` to verify that `expr` does throw an error of any type, `CHECK_THROW_AS(expr, exprtype)`/`REQUIRE_THROW_AS(expr, exprtype)` to verify that `expr` throws an exception of the type `exprtype`, or `CHECK_THROWS_WITH(expression, string or string matcher)`/`REQUIRE_THROWS_WITH(expression, string or string matcher)` to verify that `expr` throws an expression whose description matches the specified string:

```
void function_that_throws()
{
  throw std::runtime_error("error");
}

void function_no_throw()
{
}

CHECK_NOTHROW(function_no_throw());
REQUIRE_NOTHROW(function_no_throw());

CHECK_THROWS(function_that_throws());
REQUIRE_THROWS(function_that_throws());

CHECK_THROWS_AS(function_that_throws(),
                std::runtime_error);
REQUIRE_THROWS_AS(function_that_throws(),
                  std::runtime_error);

CHECK_THROWS_WITH(function_that_throws(),
                  "error");
REQUIRE_THROWS_WITH(function_that_throws(),
        Catch::Matchers::Contains("error"));
```

- Use `CHECK_THAT(value, matcher expression)`/`REQUIRE_THAT(expr, matcher expression)` to check whether the given matcher expression evaluates to `true` for the specified value:

```
std::string text = "this is an example";
CHECK_THAT(text,
  Catch::Matchers::Contains("EXAMPLE", Catch::CaseSensitive::No));
REQUIRE_THAT(text,
  Catch::Matchers::StartsWith("this") &&
  Catch::Matchers::Contains("an"));
```

- Use `FAIL (message)` to report `message` and fail the test case, `WARN (message)` to log the message without stopping the execution of the test case, and `INFO(message)` to log the message to a buffer and only report it with the next assertion that would fail.

How it works…

The `REQUIRE`/`CATCH` family of macros decompose the expression into its left- and right-hand side terms and, upon failure, report the location of the failure (source file and line), the expression, and the values on the left- and right-hand side:

```
f:\chapter11ca_03\main.cpp(19): FAILED:
  REQUIRE( a == 1 )
with expansion:
  42 == 1
```

However, these macros do not support complex expressions composed using logical operators, such as `&&` and `||`. The following example is an error:

```
REQUIRE(a < 10 || a %2 == 0); // error
```

The solution for this is to create a variable to hold the result of the expression evaluation and use it in the assertion macros. In this case, however, the ability to print the expansion of the elements of the expression is lost:

```
auto expr = a < 10 || a % 2 == 0;
REQUIRE(expr);
```

Special handling is provided to floating point values. The framework provides a class called `Approx`; it overloads the equality/inequality and comparison operators with values through which a `double` value can be constructed. The margin by which the two values can either differ or be considered equal can be specified as a percentage of the given value. This is set using the member function `epsilon()`. The value must be between 0 and 1 (for example, the value of 0.05 is 5 percent). The default value of epsilon is set to `std::numeric_limits<float>::epsilon()*100`.

Two sets of assertions, namely `CHECK_THAT`/`REQUIRE_THAT` and `CHECK_THROWS_WITH`/`REQUIRE_THROWS_WITH` , work with matchers. Matchers are extensible and composable components which perform value matching. The framework provides several matchers for strings (such as `StartsWith`, `EndsWith`, `Contains`, or `Equal`) and for `std::vector` (`Contains`, `VectorContains` and `Equal`).

The difference between `Contains()` and `VectorContains()` is that `Contains()` searches for a vector in another vector and `VectorContains()` searches for a single element inside a vector.

You can create your own matchers, either to extend the existing framework capabilities or to work with your own types. There are two things that are necessary:

1. A matcher class derived from `Catch::MatcherBase<T>`, where `T` is the type being compared. There are two virtual functions that must be overridden: `match()` which takes a value to match and returns a Boolean indicating whether the match was successful, and `describe()` which takes no arguments but returns a string describing the matcher.
2. A builder function that is called from the test code.

The following example defines a matcher for the `point3d` class, which we have seen

throughout the chapter, to check whether a given 3D point lies on a line in the three-dimensional space:

```
class OnTheLine : public Catch::MatcherBase<point3d>
{
  point3d const p1;
  point3d const p2;
public:
  OnTheLine(point3d const & p1, point3d const & p2):
    p1(p1), p2(p2)
  {}

  virtual bool match(point3d const & p) const override
  {
    auto rx = p2.x() - p1.x() != 0 ?
              (p.x() - p1.x()) / (p2.x() - p1.x()) : 0;
    auto ry = p2.y() - p1.y() != 0 ?
              (p.y() - p1.y()) / (p2.y() - p1.y()) : 0;
    auto rz = p2.z() - p1.z() != 0 ?
              (p.z() - p1.z()) / (p2.z() - p1.z()) : 0;

    return
      Approx(rx).epsilon(0.01) == ry &&
      Approx(ry).epsilon(0.01) == rz;
  }
protected:
  virtual std::string describe() const
  {
    std::ostringstream ss;
    ss << "on the line between " << p1 << " and " << p2;
    return ss.str();
  }
};

inline OnTheLine IsOnTheLine(point3d const & p1,
                             point3d const & p2)
{
  return OnTheLine {p1, p2};
}
```

The following test case contains an example on how to use this custom matcher:

```
TEST_CASE("matchers")
{
  SECTION("point origin")
  {
    point3d p { 2,2,2 };
    REQUIRE_THAT(p, IsOnTheLine(point3d{ 0,0,0 },
                                point3d{ 3,3,3 }));
  }
}
```

See also

- *Writing and invoking tests with Catch*

Controlling output with Catch

As with other testing frameworks discussed in this book, Catch reports the results of a test program execution in a human-readable format to the `stdout` standard stream. Additional options are supported, such as reporting using XML format or writing to a file. In this recipe, we will look at the main options available for controlling the output when using Catch.

Getting ready

To exemplify the way the test program's execution output could be modified, use the following test cases:

```
    TEST_CASE("case1")
    {
      SECTION("function1")
      {
        REQUIRE(true);
      }
    }

    TEST_CASE("case2")
    {
      SECTION("function2")
      {
        REQUIRE(false);
      }
    }
```

The output of running these two test cases is as follows:

```
-------------------------------------------------------
case2
  function2
-------------------------------------------------------
f:\chapter11ca_04\main.cpp(14)
.......................................................
f:\chapter11ca_04\main.cpp(16): FAILED:
  REQUIRE( false )
=======================================================
test cases: 2 | 1 passed | 1 failed
assertions: 2 | 1 passed | 1 failed
```

How to do it…

To control the output of a test program when using Catch, you can:

- Use the command-line argument `-r` or `--reporter <reporter>` to specify the reporter used to format and structure the results. Default options supplied with the framework are `console`, `compact`, `xml`, and `junit`:

```
chapter11ca_04.exe -r junit

<?xml version="1.0" encoding="UTF-8"?>
<testsuites>
  <testsuite name="chapter11ca_04.exe" errors="0"
             failures="1"
             tests="2" hostname="tbd"
             time="0.002039"
             timestamp="2017-03-02T21:17:04Z">
    <testcase classname="case1" name="function1"
              time="0.00016"/>
    <testcase classname="case2"
              name="function2" time="0.00024">
      <failure message="false" type="REQUIRE">
        at f:\chapter11ca_04\main.cpp(16)
      </failure>
    </testcase>
    <system-out/>
    <system-err/>
  </testsuite>
</testsuites>
```

- Use the command-line argument `-s` or `--success` to display results of successful test cases too:

```
chapter11ca_04.exe -s

-------------------------------------------------
case1
  function1
-------------------------------------------------
f:\chapter11ca_04\main.cpp(6)
.................................................
f:\chapter11ca_04\main.cpp(8):
PASSED:
  REQUIRE( true )
-------------------------------------------------
case2
  function2
-------------------------------------------------
f:\chapter11ca_04\main.cpp(14)
.................................................
f:\chapter11ca_04\main.cpp(16):
FAILED:
  REQUIRE( false )
=================================================
test cases: 2 | 1 passed | 1 failed
assertions: 2 | 1 passed | 1 failed
```

- Use the command-line argument `-o` or `--out <filename>` to send all of the output to a file instead of the standard stream:

```
chapter11ca_04.exe -o test_report.log
```

- Use the command-line argument `-d` or `--durations <yes/no>` to display the time,

expressed in milliseconds, that it takes each test case to execute:

```
chapter11ca_04.exe -d yes

0.000137 s: scenario1
0.000926 s: case1
-----------------------------------------------
case2
   scenario2
-----------------------------------------------
f:\chapter11ca_04\main.cpp(14)
...............................................
f:\chapter11ca_04\main.cpp(16):
FAILED:
  REQUIRE( false )

0.019106 s: scenario2
0 s: case2
4.9e-05 s: case2
===============================================
test cases: 2 | 1 passed | 1 failed
assertions: 2 | 1 passed | 1 failed
```

How it works…

Apart from the human-readable format used, by default, for reporting the results of the test program execution, the Catch framework supports two XML formats:

- A Catch-specific XML format (specified with `-r xml`).
- A JUnit-like XML format, following the structure of the JUnit ANT task (specified with `-r junit`).

The former reporter streams the XML content as unit tests are executed and results are available. It can be used as input to an XSLT transformation to generate an HTML report for the instance. The latter reporter needs to gather all of the program execution data in order to structure the report before printing it. The JUnit XML format is useful for being consumed by third-party tools, such as continuous integration server.

Several additional reporters are provided but as separate downloads. They need to be pulled into the project and explicitly included into the source code of the test program (all the headers of the additional reporters have the name format as `catch_reporter_*.hpp`). These additional available reporters are:

- *TeamCity* reporter (specified with `-r teamcity`) writes TeamCity service messages to the standard output stream. It is suitable only for integration with TeamCity. It is a streamed reporter; data is written as it is available.
- *Automake* reporter (specified with `-r automake`) writes the meta tags expected by automake via make check.
- *Test Anything Protocol* (short for, *TAP*) reporter (specified with `-r tap`).

The following example shows how to include the TeamCity header file in order to produce the report using the TeamCity reporter:

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include "catch_reporter_teamcity.hpp"
```

The default target of the test report is the standard stream `strout` (even data written explicitly to `stderr` ends up being redirected to `stdout`). However, it is possible that the output is written to a file instead. These formatting options can be combined. For instance, the next command specifies that the report should use the JUnit XML format and be saved to a file called `test_report.xml`:

```
chapter11ca_04.exe -r junit -o test_report.xml
```

See also

- *Getting started with Catch*
- *Writing and invoking tests with Catch*

Bibliography

Websites

- C++ reference http://en.cppreference.com/w/
- ISO C++ https://isocpp.org/
- More C++ Idioms https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms
- Boost http://www.boost.org/
- Catch https://github.com/philsquared/Catch
- Google Test https://github.com/google/googletest

Articles and books

- David Abrahams, 2001. *Lessons Learned from Specifying Exception-Safety for the C++ Standard Library* http://www.boost.org/community/exception_safety.html
- Michael Afanasiev, 2016. *Combining Static and Dynamic Polymorphism with C++ Mixin classes* https://michael-afanasiev.github.io/2016/08/03/Combining-Static-and-Dynamic-Polymorphism-with-C++-Template-Mixins.html
- Alex Allain, 2011. *Constexpr - Generalized Constant Expressions in C++11* http://www.cprogramming.com/c++11/c++11-compile-time-processing-with-constexpr.html
- Matthew H. Austern, 2001. *The Standard Librarian: Defining a Facet* http://www.drdobbs.com/the-standard-librarian-defining-a-facet/184403785
- Thomas Badie, 2012. *C++11: A generic Singleton* http://enki-tech.blogspot.ro/2012/08/c11-generic-singleton.html
- Eli Bendersky, 2016. *The promises and challenges of std::async task-based parallelism in C++11* http://eli.thegreenplace.net/2016/the-promises-and-challenges-of-stdasync-task-based-parallelism-in-c11/
- Eli Bendersky, 2011. *The Curiously Recurring Template Pattern in C++* http://eli.thegreenplace.net/2011/05/17/the-curiously-recurring-template-pattern-in-c
- Joshua Bloch, 2008. *Effective Java (2nd Edition)* Addison-Wesley
- Fernando Luis Cacciola Carballal, 2007. *Boost.Optional* http://www.boost.org/doc/libs/1_63_0/libs/optional/doc/html/index.html
- Bruce Dawson, 2012. *Comparing Floating Point Numbers, 2012 Edition* https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/
- Kent Fagerjord. 2016. *How to build Boost 1.62 with Visual Studio 2015* https://studiofreya.com/2016/09/29/how-to-build-boost-1-62-with-visual-studio-2015/
- Eric Friedman and Itay Maman, 2003. *Boost.Variant* http://www.boost.org/doc/libs/1_63_0/doc/html/variant.html
- Wilfried Goesgens, 2015. *Comparison: Lockless programming with atomics in C++ 11 vs. mutex and RW-locks* https://www.arangodb.com/2015/02/comparing-atomic-mutex-rwlocks/
- Kevlin Henney, 2001. *Boost.Any* http://www.boost.org/doc/libs/1_63_0/doc/html/any.html
- Howard Hinnant, *(library on GitHub)* https://github.com/HowardHinnant/date
- Nicolai M. Josuttis 2012. *The C++ Standard Library: Utilities* http://www.informit.com/articles/article.aspx?p=1881386&seqNum=2
- Nicolai Josutis, 2012. *The C++ Standard Library, 2nd Edition* Addison Wesley
- Danny Kalev, 2012. *Using constexpr to Improve Security, Performance and Encapsulation in C++* http://blog.smartbear.com/c-plus-plus/using-constexpr-to-improve-security-performance-and-encapsulation-in-c/
- Danny Kalev, 2012. *C++11 Tutorial: Introducing the Move Constructor and the Move Assignment Operator* http://blog.smartbear.com/c-plus-plus/c11-tutorial-introducing-the-move-constructor-and-the-move-assignment-operator/
- David Kieras, 2013. *Why std::binary_search of std::list Works, But You Shouldn't Use It!* EECS 381
- Matt Kline 2017. *Comparing Floating-Point Numbers Is Tricky*, http://bitbashing.io/compari

ng-floats.html
- Andrzej Krzemienski, 2016. *Another polymorphism* https://akrzemi1.wordpress.com/2016/02/27/another-polymorphism/
- Andrzej Krzemienski, 2011. *Using noexcept* https://akrzemi1.wordpress.com/2011/06/10/using-noexcept/
- Andrzej Krzemienski, 2014. *noexcept—what for?* https://akrzemi1.wordpress.com/2014/04/24/noexcept-what-for/
- Andrzej Krzemienski, 2013. *noexcept destructors* https://akrzemi1.wordpress.com/2013/08/20/noexcept-destructors/
- John Maddock and Steve Cleary, 2000. *C++ Type Traits* http://www.drdobbs.com/cpp/c-type-traits/184404270
- Arne Mertz, 2016. *Modern C++ Features – constexpr* https://arne-mertz.de/2016/06/constexpr/
- Scott Meyers, 2014. *Effective Modern C++*, *O'Reilly*
- Scott Meyers and Andrei Alexandrescu, 2004. *C++ and the Perils of Double-Checked Locking* http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf
- Bartosz Milewski, 2009. *Broken promises–C++0x futures* https://bartoszmilewski.com/2009/03/03/broken-promises-c0x-futures/
- Bartosz Milewski, 2008. *Who ordered sequential consistency?* https://bartoszmilewski.com/2008/11/11/who-ordered-sequential-consistency/
- Bartosz Milewski, 2008. *C++ atomics and memory ordering* https://bartoszmilewski.com/2008/12/01/c-atomics-and-memory-ordering/
- Oliver Mueller, 2014. *Testing C++ With A New Catch* http://blog.coldflake.com/posts/Testing-C++-with-a-new-Catch/
- Ashwin Nanjappa, 2014. *How to build Boost using Visual Studio* https://codeyarns.com/2014/06/06/how-to-build-boost-using-visual-studio/
- M.E. O'Neill, 2015. *C++ Seeding Surprises* http://www.pcg-random.org/posts/cpp-seeding-surprises.html
- M.E. O'Neill, 2015. *Developing a seed_seq Alternative* http://www.pcg-random.org/posts/developing-a-seed_seq-alternative.html
- M.E. O'Neill, 2015. *Everything You Never Wanted to Know about C++'s random_device* http://www.pcg-random.org/posts/cpps-random_device.html
- M.E. O'Neill, 2015. *Simple Portable C++ Seed Entropy* http://www.pcg-random.org/posts/simple-portable-cpp-seed-entropy.html
- John Pearce, *Floating Point Numbers* http://www.cs.sjsu.edu/~pearce/modules/lectures/co/ds/floats.htm
- Jeff Preshing, 2013. *Double-Checked Locking is Fixed In C++11* http://preshing.com/20130930/double-checked-locking-is-fixed-in-cpp11/
- Rick Regan, 2010. *Hexadecimal Floating-Point Constants* http://www.exploringbinary.com/hexadecimal-floating-point-constants/
- Eugene Sadovoi, 2015. *Building and configuring boost in Visual Studio (MSBuild)* https://www.codeproject.com/Articles/882581/Building-and-configuring-boost-in-Visual-Studio-MS
- David Sankel, 2015. *A variant for the everyday Joe* http://davidsankel.com/c/a-variant-for-the-everyday-joe/
- Arpan Sen, 2010. *A quick introduction to the Google C++ Testing Framework* http://w

ww.ibm.com/developerworks/aix/library/au-googletestingframework.html
- Bjarne Stroustrup, 2000. *Standard-Library Exception Safety* Addison Wesley http://stroustrup.com/3rd_safe.pdf
- Herb Sutter, 2013. *GotW #90 Solution: Factories* https://herbsutter.com/2013/05/30/gotw-90-solution-factories/
- Herb Sutter, 2002. *A Pragmatic Look at Exception Specifications* C/C++ Users Journal, 20(7) http://www.gotw.ca/publications/mill22.htm
- Herb Sutter, 2012. *GotW #102: Exception-Safe Function Calls* https://herbsutter.com/gotw/_102/
- Herb Sutter, 2013. *My Favorite C++ 10-Liner* https://channel9.msdn.com/Events/GoingNative/2013/My-Favorite-Cpp-10-Liner
- Herb Sutter, 2001. *Virtuality*, C/C++ Users Journal, 19(9) http://www.gotw.ca/publications/mill18.htm
- Andrey Upadyshev, 2015. *PIMPL, Rule of Zero and Scott Meyers* http://oliora.github.io/2015/12/29/pimpl-and-rule-of-zero.html
- Todd Veldhuizen, 2000. *Techniques for Scientific C++* http://www.cs.indiana.edu/pub/techreports/TR542.pdf
- Baptiste Wicht, 2014. *Catch: A powerful yet simple C++ test framework* https://baptiste-wicht.com/posts/2014/07/catch-powerful-yet-simple-cpp-test-framework.html
- Anthony Williams, 2009. *Multithreading in C++0x part 7: Locking multiple mutexes without deadlock* https://www.justsoftwaresolutions.co.uk/threading/multithreading-in-c++0x-part-7-locking-multiple-mutexes.html
- Anthony Williams, 2008. *Peterson's lock with C++0x atomics* https://www.justsoftwaresolutions.co.uk/threading/petersons_lock_with_C++0x_atomics.html
- Benjamin Wolsey, 2010. *C++ facets* http://benjaminwolsey.de/node/78